Volume 23 Number 2

ACTA CYBERNETICA

Szeged, 2017

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. An electronic version of the puplished paper is provided for the authors in PDF format.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in LaTeX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: `acta@inf.u-szeged.hu`

Web access. The above information along with the contents of past issues are available at the Acta Cybernetica homepage `https://www.inf.u-szeged.hu/en/kutatas/acta-cybernetica` .

Zoltan Kato
Department of Image Processing
and Computer Graphics
Szeged, Hungary
kato@inf.u-szeged.hu

Alice Kelemenová
Institute of Computer Science
Silesian University at Opava
Opava, Czech Republic
Alica.Kelemenova@fpf.slu.cz

László Lovász
Department of Computer Science
Eötvös Loránd University
Budapest, Hungary
lovasz@cs.elte.hu

Gheorghe Păun
Institute of Mathematics of the
Romanian Academy
Bucharest, Romania
George.Paun@imar.ro

Arto Salomaa
Department of Mathematics
University of Turku
Turku, Finland
asalomaa@utu.fi

László Varga
Department of Software Technology
and Methodology
Eötvös Loránd University
Budapest, Hungary
varga@ludens.elte.hu

Heiko Vogler
Department of Computer Science
Dresden University of Technology
Dresden, Germany
Heiko.Vogler@tu-dresden.de

Gerhard J. Woeginger
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
gwoegi@win.tue.nl

Conference of PhD Students in Computer Science

Guest Editor:

Rudolf Ferenc

Department of Software Engineering
University of Szeged
Szeged, Hungary
ferenc@inf.u-szeged.hu

Preface

The *10th Jubilee Conference of PhD Students in Computer Science (CSCS)* was organized by the Institute of Informatics of the University of Szeged (SZTE) and held in Szeged, Hungary, between June 27–29, 2016.

The members of the *Scientific Committee* were the following representatives of the Hungarian doctoral schools in computer science:
Zoltán Fülöp (Co-Chair, SZTE), Lajos Rónyai (Co-Chair, SZTAKI, BME), Péter Baranyi (SZE), András Benczúr (ELTE), Hassan Charaf (BME), Tibor Csendes (SZTE), László Cser (BCE), Erzsébet Csuhaj-Varjú (ELTE), János Demetrovics (ELTE), Imre Dobos (BCE), István Fazekas (DE), János Fodor (ÓE), Aurél Galántai (ÓE), Zoltán Gingl (SZTE), Tibor Gyimóthy (SZTE), Katalin Hangos (PE), Zoltán Horváth (ELTE), Csanád Imreh (SZTE), Márk Jelasity (SZTE), Zoltán Kása (Sapientia EMTE), László Kóczy (SZE), János Levendovszki (BME), László Nyúl (SZTE), Attila Pethő (DE), Gábor Prószéki (PPKE), Tamás Szirányi (SZTAKI), Péter Szolgay (PPKE), János Sztrik (DE), János Tapolcai (BME), and János Végh (ME).

The members of the *Organizing Committee* were:
Rudolf Ferenc (Chair), Balázs Bánhelyi, Tamás Gergely, and Zoltán Kincses, and Attila Kertész.

There were more than 50 participants and 46 talks in several fields of computer science and its applications (14 sessions). The talks were going in sections in Testing, Mining, Machine Learning, Medical Solutions, Program Analysis, Programming Languages, Image Processing I., Model Matching , Image Processing II., Algorithm, Cloud/IoT, Networking, Models, and Optimization. The talks of the students were completed by 3 plenary talks of leading scientists: Dániel Marx (MTA SZTAKI), László Pyber (MTA Rényi), and Dániel Varró (BME).

The open-access scientific journal Acta Cybernetica offered PhD students to publish the paper version of their presentations after a careful selection and review process. Altogether 28 manuscripts were submitted for review out of which 16 were accepted for publication in the present special issue of Acta Cybernetica. We wish to thank David P. Curley for reviewing and correcting the papers from a linguistic point of view.

The full program of the conference, the collection of the abstracts and further information can be found at `http://www.inf.u-szeged.hu/~cscs`.

On the basis of our repeated positive experiences, the conference will be organized in the future, too. According to the present plans, the next meeting will be held around end of June 2018 in Szeged.

Rudolf Ferenc
Guest Editor

Acta Cybernetica **23** (2017) 441–470.

A Personalized Multi-Path and Multi-User Traffic Analysis and Visualization Tool

Cristian Babau[a], Marius Marcu[a], Mircea Tihu[a], Daniel Telbis,[a] and Vladimir Cretu[a]

Abstract

Traffic optimization is a subject that has become vital for the world we live in. People these days need to get from a starting point to a destination point as fast and as safe as possible. Traffic congestion plays a key role in the frustration of people and it results in lost time, reduced productivity and wasted resources. In our study we seek to address these issues by proposing a real-time road traffic planning system based on mobile context and crowd sourcing efforts. The first step toward this goal is real-time traffic characterization using data collected from mobile sensors of drivers, pedestrians, cyclists, passengers, etc.. We started developing a data collection and analysis system composed of a mobile application in order to collect user context data and a Web application to view and analyze the data. This new system will eventually give the users an automatically optimized route to the destination and predict the users' traveling route based on live traffic conditions and historical data.

Keywords: traffic optimization, mobile context, route analysis, visualization tool

1 Introduction

After the appearance of smart phones in the mobile telephones market an increasing number of applications are trying to provide specialized services for their users. A method for modifying the application's behavior is based on the context in which the mobile phone is currently functioning and the applications that use this context are called context-aware applications. Context-aware applications use the data sensed by the device sensors, such as a sound sensor, a light sensor, a temperature sensor, a movement sensor or a position sensor and they map it using certain algorithms to a real-life context (the user is inside or outside a building, the weather is cold or hot, the environment is quiet or loud); then, based on this context, different services can be provided by the application.

[a]Polytechnic University of Timisoara, E-mail: cristian.babau@student.upt.ro, mmarcu@cs.upt.ro, mirceatihu@gmail.com, daniel_telbis@yahoo.com, vcretu@cs.upt.ro

DOI: 10.14232/actacyb.23.2.2017.2

Integrating the mobile context of the drivers when analyzing urban traffic and planning transportation routes is the next step in urban intelligent transportation systems. The key goal of this study is to collect traffic data from users using their mobile devices and then to aggregate the raw data obtained in order to monitor, analyze the data and also characterize route segments and generate route predictions. In this study we present a mobile Android application designed for sample data from the built-in sensors and a Web application used to visualize and analyze the data sampled from mobile devices together with the conclusions resulting from using the applications. The visualization tool is public and it can be accessed at http://mobilebackend.cs.upt.ro/sensor-data-viewer/overview/index.php.

The Android application uses a service that runs in the background and collects sensor data from the most common sensors encountered in the mobile devices, namely an accelerometer, gyroscope, magnetometer, location from the network provider and GPS, linear acceleration, proximity, sound and light levels. The collected data is then stored in a local database, it can be sent on to a remote database on a remote server. The source code of the mobile client is open and it is published on bitbucket.org/lipan19/disertatie-v2.

The Web application uses both aggregated and individual data got from the remote sensor to help users visualize and analyze their collected data. All the locations collected are then displayed on a Google Map so the user can see his everyday routes and for each route he can analyze the speed, acceleration, light level, sound level for different parts of the journey. By splitting each part into atomic segments and performing segment profiling from all the data gathered for a particular journey, an accurate representation of the segments can be constructed, so the user can monitor his own driving habits. Based on this representation the user can later estimate and characterize a whole trip from one point to another by reconstructing the journey from the segments. The source code of the Web application is open and it is published on bitbucket.org/lipan19/sensor-data-viewer.

The paper is structured as follows. In Section II there we discuss the related work that has been carried out in the field. In section IV, we introduce our solution overview to the problem. Next, in section V we present the results of our experiments. Lastly, in Section VI we summarize our key conclusions and make some suggestions for future research.

2 Related work

Traffic advisory systems were introduced fairly recently to help people plan their routes and optimize their traveling distance and time. These systems are called Intelligent Transportation Systems or ITS for short [1]. ITS applies advanced communication, information and electronics technology to help solve transportation problems like traffic congestion, safety, transport efficiency and environmental conservation. Several aspects have to be addressed in any modern ITS, as suggested in Figure 1.

Traffic information data collection is one of the most important aspects of ITS

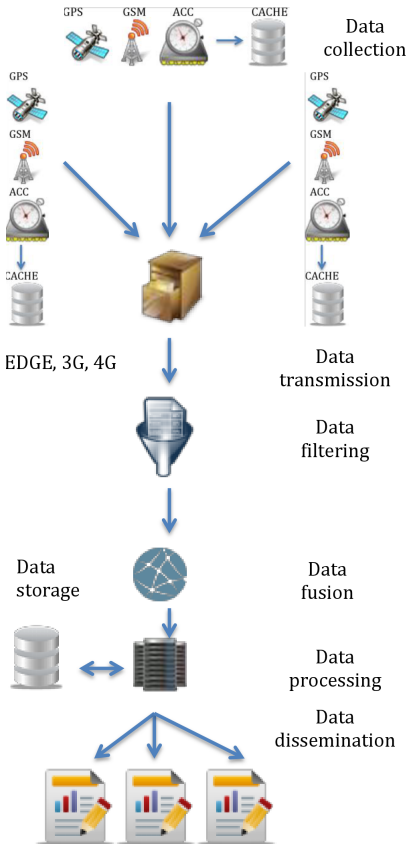


Figure 1: ITS overview

and the first that needs to be addressed when developing such systems. There are many ways that traffic data could be collected and subsequently analyzed and processed by traffic data analysts and automated systems. These methods can be grouped into two general categories depending on the source of the data: the first type involves colleting data from dedicated and reliable stations (both fixed stations and mobile agents) and the second type involves collecting data from the users participating in the traffic (e.g. crowd sourcing). The main role of data collection is gathering primary information about road traffic conditions in order to produce more complex knowledge, including traffic congestion, incidents that have occurred, the state of a section of the route they are traveling on and the weather conditions.

Most recent solutions use the mobile devices of the participants while using different navigation or context-based mobile applications. These sensors move freely with the vehicles and collect traffic data. A mobile sensor network of traffic participants must be able to accomplish three major tasks [1]:

- Location determination it must be able to determine the coordinates of the probe to same degree of accuracy;
- Map matching it must be able to pinpoint the coordinates of the probe in the road network to some degree of accuracy;
- Evaluation of the of traffic information the system must be able to process the data collected and generate traffic reports.

Google Traffic uses location and speed data collected from the users who have Android devices in order to provide information about traffic jams and the best routes in a specified area. The information is displayed on a Google Map, where the best route segments are shown in green, while the slow segments are shown in red. All the data collected is anonymous, and the sum of the data collected from more devices gives the final value of the segment, which is just characterized by its color, no other metric like speed or acceleration being provided. The service provides two operation modes, namely one in which the live-traffic conditions are displayed from data harvested in real-time from the users, and an hour-specific traffic, which displays data from the usual traffic for the given day of the week, for a particular time frame, between 06:00 and 22:00.

Waze is a platform which provides a mobile application that collects data from the users' smart devices and transfers this data to the application servers, where they are used to detect the roads where the traffic is congested and display it in real time. The users can also add traffic events to the mobile application system like accidents, roads under construction and closed roads. The platform also provides a Web application where the users can monitor on a map all the events added by other users and also note the most traffic-congested roads, while also having an interface to determine the fastest route between two points, the issue with the estimates often being that they are far too optimistic for the actual traffic.

In [2], the authors created an application that attempts to reconstruct and display the continuous traffic flow on different routes based on the data collected by sets of road sensors which measure the distances and speeds. The algorithm estimates the full state of the traffic flow by using statistical inference methods and being able to reconstruct a 3D version of the traffic at a particular date and hour on the particular routes, the main focus of the application being the highway traffic. EnviroCar [3] is a platform composed of a mobile application and a Web application which, using an On-Board Diagnosis adapter for cars, gathers car and road data like carbon dioxide emissions, noise emissions and when cars are stationary, which are automatically recorded in the mobile application. In order for the user to gather data, he needs to attach the On-Board Diagnosis adapter to his car, install the enviroCar application on his mobile device and let the adapter transmit the data. After gathering the data, it is uploaded to a remote server where it can then be analyzed using the Web application, which displays on a map the areas with high carbon dioxide emissions, high noise or long stationary times in the traffic.

Another approach for traffic prediction is outlined in [4], where the authors develop an application that allows one to analyze and predict the traffic jams on

the main routes in North America. The application used the Microsoft JamBayes research project, which was started to permit the prediction of the traffic flow on the North American roads based on data gathered over a period of five years from different vehicle sources and it has an interface which displays the traffic flow in a color-coded way, green roads signifying a lighter flow, while red roads signify that a heavy flow traffic in present. By using screenshots taken from the JamBayes project interface, the authors perform an image recognition in order to determine the areas where traffic jams are present and the system displays them as a heat-map layer over a Bing map so the users can note which routes they should avoid.

Traffic Origins [5] is a visualization application created for the traffic management controllers in order to monitor and analyze the changes in traffic affected by accidents or other road transit events. The application allows the user to observe the routes during certain time frames before, during and after the traffic events occurred. A map is displayed where all the routes are color-coded from green to red, depending on the level of congestion for each particular segment. To observe the changes caused by an accident, a circle will appear on the map just prior to the event and it will remain on the map for a certain time after the event has occurred. This way, the traffic manager can observe how the conditions are changing during an traffic event and decide how best to handle it.

To address practical problems in ITS, a suite of schemes and tools are required for realizing and analyzing big data in social transportation systems, among which of paramount importance is visual analysis [6]. Visual analytics is the science of analytic reasoning supported by interactive visual interfaces. In the core of visual analytics is data visualization, which transforms various types of data into appropriate visual representations, and greatly improves the efficiency of data comprehension and analysis [6]. Powerful tools are needed to enhance the user experience and proficiency in manipulating and understanding huge amounts of data collected from many users in various conditions and qualities. One such tool is the Iris geographic information system (GIS) - based a platform using the smartphone Android default API for geolocation [7]. The server side of the Iris platform complements the mobile system to collect data by storing, processing, analyzing, managing and presenting the results. Additional metrics related to the travel time and vehicle's speeds contribute to the assessment of traffic management issues [7].

In [8], the author presents a Web application which uses live traffic congestion data from Google Maps traffic layer for real-time congestion calculation. This application focuses in particular on urban roads congestion analysis. Besides the spatial aspect, the study concentrates on the temporal dimension of road link traffic by providing stored-time-of-the-day analysis data of when the congestion level was highest [8].

The most common method used to estimate the possible routes between two points is Dijkstra's algorithm, which was created for finding the shortest paths between different nodes in a graph, after supplying the initial node and the destination node. The algorithm starts from the initial node and advances through the graph, while keeping the distance count for each route until it reaches the destination node. To estimate the routes between two points on the map, a modified version

of Dijkstra's algorithm can be used, which instead of starting from the initial point and going to the destination point, starts from both the start and end points and expands the routes on both sides until the routes meet in the middle. This way, the computational time is reduced because the operations can be performed in parallel. Depending on the distance between the start and end point, the algorithm can use street layers to avoid taking into account small streets for routes traveling from a city to another; the algorithm can use a layer for the highways, one for European roads, national roads and so on [9].

The IRMA project [10] seeks to implement a software framework that targets personal mobility that is green oriented, shared and public transportation and provide an Integrated Real-time Mobility Assistant. IRMA is an extendible modular platform using multiple sources of data (crowd, open, social, and sensor data) and it provides traffic services to several stakeholders, which includes the municipality, citizens, and transport providers. To access data, a service-oriented architecture based on an event driven platform has been adopted.

In [11] the authors provide a comprehensive survey on the state-of-the-art in visualization techniques for large-scale vehicle traffic data. Existing techniques include conventional ones such as 2D and 3D maps or more innovative ones, such as geometric projections (which display projections of multidimensional data sets), pixel-oriented views (which map each data value to a colored pixel) and hierarchical presentations.

The next step in ITS infrastructures is based on space-air-ground data, as introduced in [12]. The authors examine the new key data and technologies for ITS, naming satellites data and helicopter collected traffic data. The future approaches on ITS will probably include space-air-ground data acquisition sensors, dynamic data transmission, massive data storage, multi-source data fusion, massive data mining and analysis.

In contrast to existing solutions, we provide a complete system for collecting and analyzing urban traffic data. On our data, many specific and detailed algorithms or metrics can be obtained, in contrast to the limited data provided by Google Traffic or Waze. The main difference is that our work seeks to provide open traffic data augmented with contextual information. The present paper presents the basic low-level infrastructure we will use to analyze and label data gathered from the users.

On the client side, there are multiple applications in which the context-awareness is used to provide personalized services for the users, each researcher trying to develop new ways of determining the context while minimizing the battery consumption. Several related applications and techniques used to improve the energy efficiency of mobile context-aware data gathering applications are discussed below.

In [13], a model is proposed that is based on the user's context provides appropriate security for different applications The authors think that the same user, in different contexts, should have different roles and different levels of permissions in the applications, since some environments are considered safer for accessing certain kind of data than others. By mapping the raw data received from various sensors from the mobile device to the user's context, the authors created three classes of

roles (R1, R2, R3), each of them having different degrees of permission for different actions, depending on the level of privacy required, each user having then assigned a different role based on his context. Three types of locations are defined, each with three sub-types, which are mapped to the role classes that determine how crowded the environment is: indoor, with the classroom(R2, R3), quiet room(R3, R3) and market hall(R1, R2) sub-types, outdoor, with a square(R2, R3), park(R2, R3) and street(R1, R2) sub-types and transportation with a bus(R1, R2), the metro(R1, R2) and a train(R1, R3) sub-types, where the first value represents that for a high crowd, and the second value that for a small crowd and R1 <R2 <R3, R3 granting permission the most often.

Seeking to minimize the energy consumption when detecting and processing the context, a new approach is proposed in [14] that modifies the general way of processing the context data. The general flow starts from the sensors, through the pre-processing feature extraction and context recognition, the context change being detected at the final step. The authors attempt to identify the change of context earlier in this flow, based on a lower level query, to avoid the processing of more computationally intensive procedures like decision tree logic. A middle-tier framework, called SeeMon was implemented, to provide an easier way to map certain actions to certain contexts. SeeMon provides a context monitoring query, which permits the user to map an action with a certain duration when a certain context is active, by using the following format: `Context <context element> ALARM <type> DURATION <duration>`. For the context element the equality and inequality operations are supported, while there are two alarm types an instant one that is triggered when the context conditions are met and a timed one, that will be triggered after a specified time once the context conditions are met. An alarm will be triggered if all the context conditions are met and the `<context element>` value changes from false to true, the duration parameter specifying how long the query should run.

In [15], the authors carried out an experiment to measure which sensors are the most energy efficient and which are consume the most energy; after completing the experiment and determining the most energy-efficient sensors, they implemented a system called SenseLess that relies heavily on these sensors, while using the most energy-consuming ones as little as possible and thus increase the battery lifetime.

The energy used for sampling the sensors and sending the data to a server, together with the actual cost of the transfer operation is also discussed in [16]. The author proposes a solution that leverages the costs for non-data-plan users, while also mining the energy consumption cost for data-plan users, by using WiFi or Bluetooth transfer to send the sensed data, saving 45%-50% in the data cost and 55%-65% in the energy consumption for each type of user.

3 Solution overview

The architecture presented in Figure 2 is composed of modules that represent the main characteristics and functionality of the system. To determine the user context

changes and to measure the energy efficiency of the operation, we first needed to gather actual data from mobile devices used in different environments, then process and analyze this data and identify patterns corresponding to each user context. To achieve this, we implemented a mobile application that was used for data sampling, and it also collected input concerning the current context.

To analyze the data collected and aggregated, a Web application was implemented, that was used to visualize the user routes, calculate the distances and user speed and also analyze the data collected, which was plotted by the application. Based on all the data sampled by the sensors, metrics are then computed for each segment and stored in order to characterize each segment from the city by its speed, acceleration and sound level, all the data being displayed on the map so as to have an overview of an entire area.

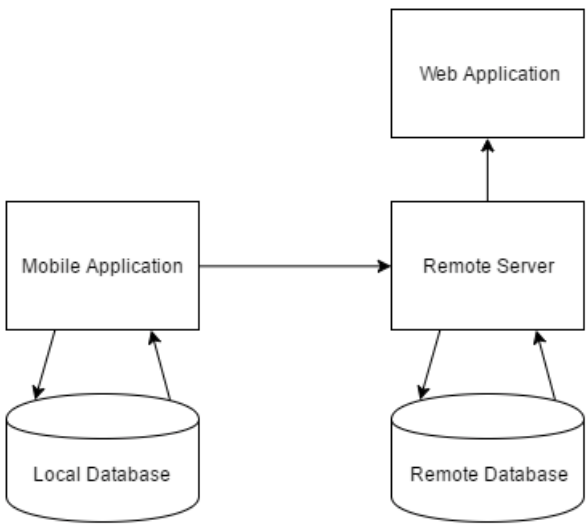


Figure 2: Solution overview

3.1 Our Mobile Data Sampling Application

In order to collect data, we created an Android application that will determine which sensors are available on the current device, the user being able to configure in the application which sensors should be enabled and how often they should be queried. The user can enable or disable sensors and also configure the querying period. By default, the following sensors are enabled with the following sampling time resolutions, as presented in table 1.

For the mobile application, the goal was to collect raw sensor data in a reliable way and to study the impact on the power usage. It is known that the GPS has a higher energy consumption compared to other device sensors. The mobile

Table 1: Sensor default sampling rates.

Sensor	Time (milliseconds)
Accelerometer	100
Gyroscope	100
Magnetometer	100
Linear Acceleration	100
Proximity sensor	1000
Light sensor	1000
Barometer	1000
Sound level	1000
Foreground application	2000
Location	5000

application will be used to determine the erxtent of its impact on the battery life is.

Another goal of the mobile application is to collect location data (especially from the GPS) in a more efficient manner by turning off location sampling if the device is not moving.

The measuring of the battery usage the internal Android battery usage monitor was performed and also a third party application called GSAM Battery Monitor, that also relies on the internal battery monitor. The data provided by the battery monitor displays the battery usage for the application, for the GPS and for the Google Play Service. In order to determine the total battery consumption, these need to be added together.

The aim of these measurements is not to give an absolute result that should be used as a reference for other applications and devices. The main purpose of these measurements is to compare the different location gathering methods with each other under similar conditions. Other measurements focused on determining the impact of sensor data collection, network location collection and sound level collection.

In order to be able to receive network location data, the device had to have an active SIM card and also be connected to the network at all times. Because the test must be performed under similar conditions, the network connection was left active during measurements that did not require it. The accuracy of the measurements changed slighty because of the device usage for activities such as phone calls and messaging, but these were kept to a minimum for the duration of the measurements. In all the tests the device was fully charged, with the configurations already set. After this, the data collection service was started. An alarm was set in order to notify the user to stop the process and save the results, this having a minimal impact on the results.

Three kinds of tests were run in order to estimate the efficiency of the sensor

readings within the mobile client during normal usage conditions. The internal battery monitor provides the usage percentages relative to the percentage of the battery used. In order to display the battery consumption relative to the total battery the results had to be normalized. Energy accounting focused on built-in sensors, sound level and network location. The percentages of these measurements are of course relative to the total battery level divided by the power used during these measurements. The first test configuration focuses on the main built-in sensors (accelerometer, gyroscope, light sensor, proximity sensor, pressure sensor and magnetic field) used altogether at default sampling rates shown in table 1. Network location and noise level are activated separately during the second test and third test, respectively.

Table 2 presents the results of the six hour tests executions repeated three times, with the previous mentioned configurations. The total battery energy used indicates the amount of battery energy consumed by the device overall. The percentages for battery energy consumed by the application are relative to the total available energy, not the total energy used.

Table 2: User application battery usage.

Test type	Total battery used by system	Fraction of the energy used by app	CPU time
Built-in sensors	11.5%	0.43%	55.5s
Network location	14%	0.14%	17.5s
Sound level	7.5%	12.5%	24.5s

The second set of tests were run in order to measure the impact on the battery consumption using several location gathering methods: Google services with high accuracy priority (GS-HA); Google services with balanced priority (GS-BP); a dynamic location sampling algorithm (Dynamic); and basic location sampling (Basic). The energy consumption while using the dynamic collection mode depends on the amount of actual movement type.

Table 3: Location sampling battery usage.

Test type	Total battery used by system	Fraction of the energy used by app	CPU time
GS-HA	91.5%	57%	5039s
GS-BP	39%	3%	357s
Dynamic	29%	16%	541s
Basic	84.5%	44%	1991s

The results in table 3 are based on the battery profiling tool which gives the

percentages for each application and system service. The total battery energy consumed by the application is the sum of energy consumption for the data collection application, the GPS location service and when the Google Play Service is used.

3.2 A Personalized Multi-Path Analyzer

The data gathered from all the devices that have installed the data collection application are stored in a database to be used afterwards by computing it different metrics for the different routes used and also determine different patterns for each user. The database contains an individual table for each sensor queried and uploaded by mobile clients.

In addition to the data collected from the default device sensors, we query data related to the application that is currently active on the device and the battery status, together with the context data entered by the user in the application, hence three extra tables were created for this.

The battery level data table stores the ID of the device from which the data was collected, the timestamp and the energy level of the battery at the given time. The contexts table stores the ID of the device from which the data was collected, the timestamp and a string representing the context entered by the user in the data-collecting application. Along with to all these tables, an extra devices table was added to store the ID of each device and the name of the users using it.

A Web application was created that helps the user analyze all the data gathered by the mobile sampling application and it allows him to visualize the locations from each particular route on a map based on the Google Maps JavaScript API and also analyze route segments, splitting the route into atomic parts with the Google Maps Directions API and then recomposing new routes for which we calculate different metrics. The personalized multipath analyzer consists of of three main tools, which are presented later on. The Web application is described in the next section.

4 Traffic Visualization Tool

4.1 Visualization of raw data

The first tool we implemented was the visualization tool, which allows the user to visualize on a Google Map the route used by the user when moving from the start point to the end point based on the collected locations, while also allowing the user to analyze the data gathered from all the sensors.

Before starting the visualization tool, we need to choose which user we want to analyze the routes and the data, after which two select boxes will be displayed, one is called Start time and the other is called Stop time, each one containing the context start and end events introduced by the user in the mobile application. The options in the two select boxes display the date and hour when the event started or ended and the current context, for example: 2016.05.27 08:34:13 start-car, 2016.05.27 08:41:38 car-stop, 2016.03.27 08:34:13 start-pedestrian, 2016.03.27 08:41:38 pedestrian-stop. When the user chooses a start event from the Start

time select box, the corresponding event from the Stop time select box will be automatically selected by the application. After the user has selected the desired start and stop time we use their timestamps to detect all the locations that have been stored for that user between the two timestamps and we display a marker on the map for each location so we can observe the route the user used.

In addition, we also make a query to the Google Maps Directions API and we can also display on the map the best route suggested by the API, together with some metrics so we can compare the actual route with the one recommended by the API: we display the distance between the start and stop location, the time estimated by the API, the actual time that passed while travelling from the start point to the end point, the average speed estimated by the API and the actual average speed measured by the device sensors, like that shown in Figure 3.

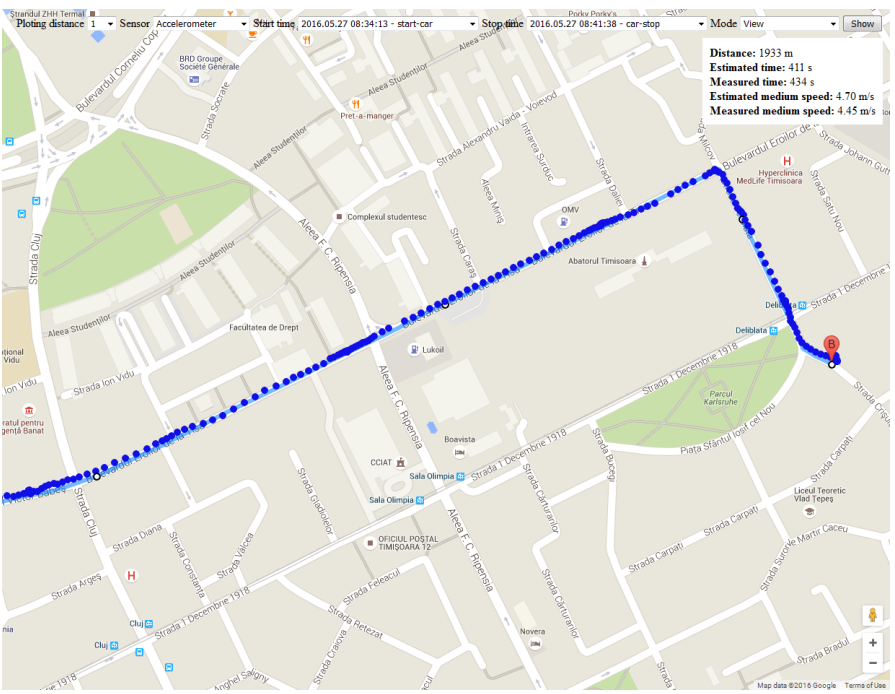


Figure 3: Visualization of the raw data

Having got the timestamp corresponding to the start time and the timestamp corresponding to the stop time we can also query all the data stored in the database for all the sensors between the two timestamps and we can plot this data on a graph over the map so the user can analyze patterns on the different routes. We added two extra select boxes, a sensor decide box and a plotting distance select box. From the sensor select box the user can select which sensor he wants to analyze, then the application will plot the values for that particular sensor on the graph. The second select box was because for long routes the amount of data that should be

displayed on the graph is too large and also impossible to analyze this way, so the user can select from the box how many distance-points from the selected point the plotted graph should display. The available options for the plotting distance are: 1, 2, 3, 5, 10, 20 and all. After the user has selected the desired sensor and the desired plotting distance he can hover over the desired location point on the map and the data corresponding to the plotting range and that point in the middle will be displayed on the screen.

There are two types of plots that can be analyzed, namely a single plot for the sensors that retrive single values, as shown in the Figure 4, like the light sensor, the pressure sensor, the proximity sensor and for the sound and the battery level values stored. The graph displays all the data existing in the database for the current selected user between the timestamp corresponding to the start location and the timestamp corresponding to the stop location, when hovering over one particular location on the graph the value for that location being displayed for the user. The second type of plot is a multiple graph of sensor values (see below)

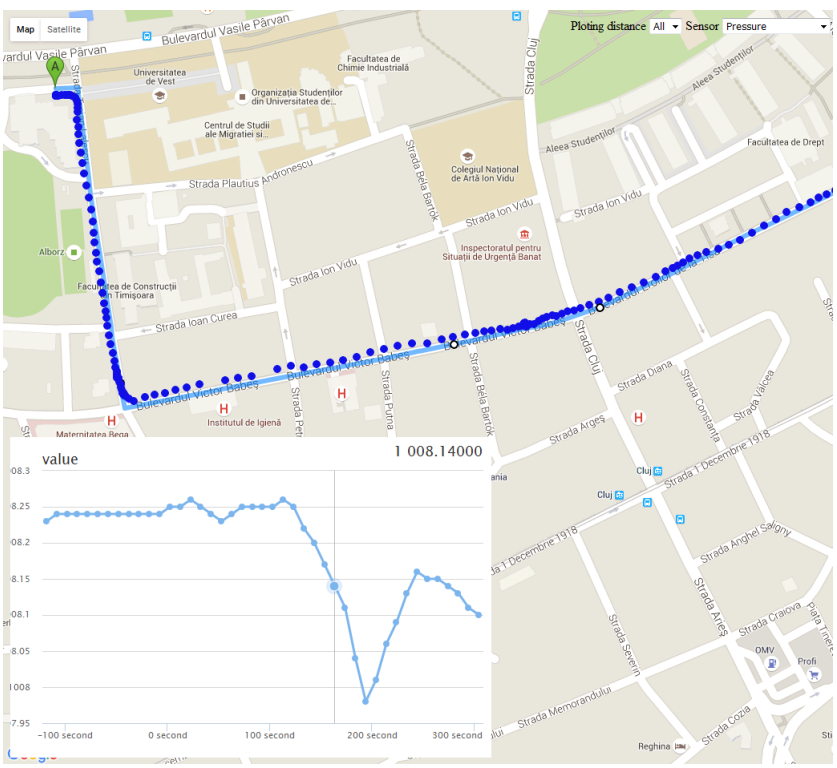


Figure 4: Visualization details of single sensor data

For the accelerometer, the gyroscope, the linear acceleration and the magnetic field a three-plot graph is used because each of these sensors return a set of data

with three components, one for each axis, so we need to plot them accordingly so the user can visualize all this data from the data set the same graph. Here we display three interconnected graph, while hovering a point from one graph corresponding to one of the axes, and the corresponding values for the other two axes are also displayed so they can be analyzed together by the user, like that seen in Figure 5.

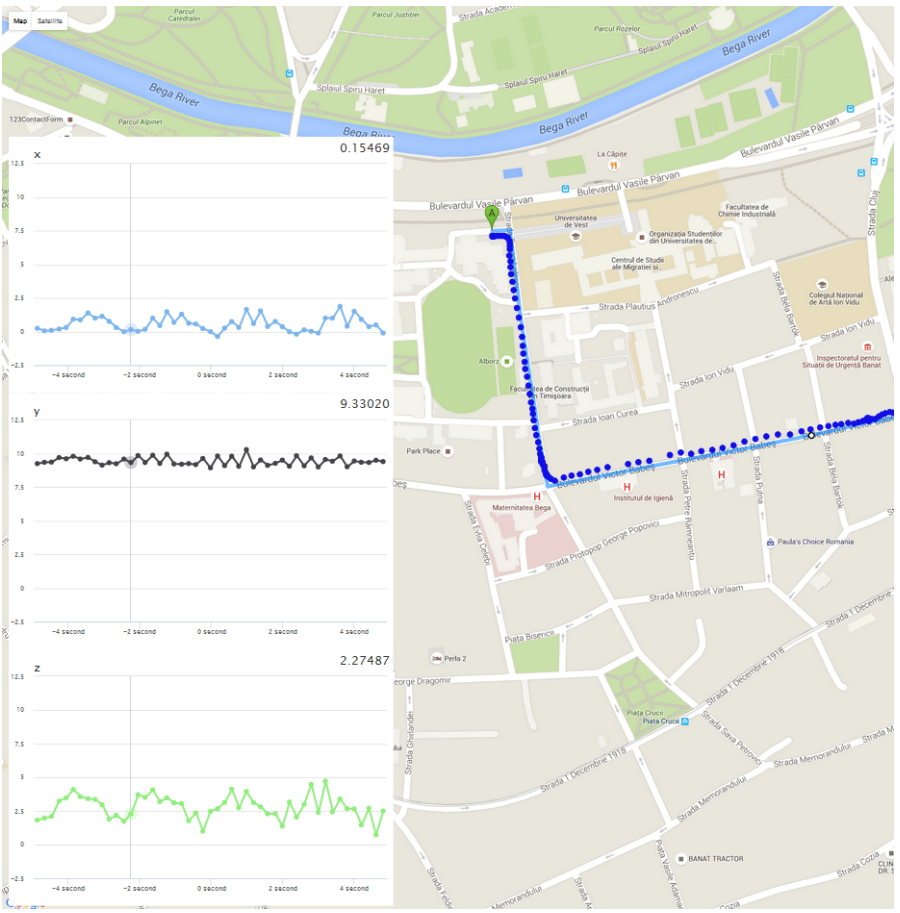


Figure 5: Visualization details of multiple axis sensor data

4.2 Segmentation Tool

The second tool that can be used in the analysis application is the segmentation tool, which performs a map matching between the route taken by the user and the route provided by the Google Maps Directions API, characterizing each segment from the route based on the data previously collected by the mobile application.

The segmentation tool also provides two select boxes: a Start time select box and a Stop time select box, that allow the user to decide which route he wants to process. After selecting a route, the user can start the process by clicking the Process segments button and he will have to wait until the progress bar displayed near the button is full so he can see that the entire process is completed and all the data has been stored in the database.

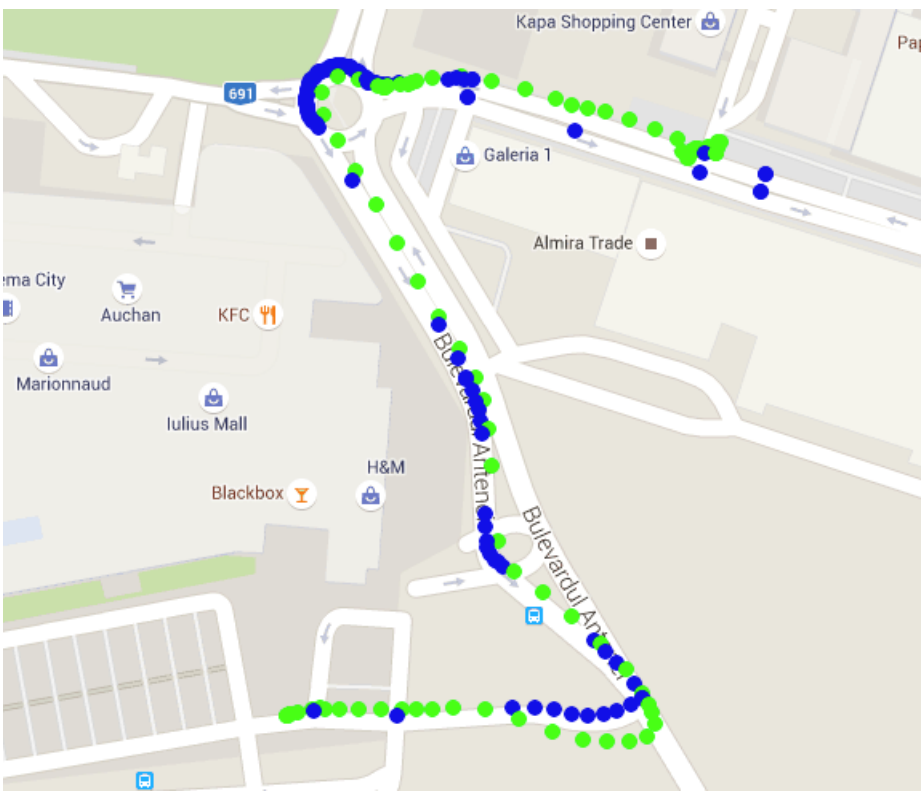


Figure 6: Visualization of segment limits

The segmentation process starts by sending a request to the Google Maps Directions API containing the start location, the end location and a number of eight waypoints, which are points from along the route followed by the user, the provided response being then used to map the data collected from the mobile application to the route segments. The response from the Directions API contains an array of locations that are the same if the user uses a certain road, regardless of the starting point or ending point of the route; anytime a user drives along a certain road the response from the Google Maps Directions API will always contain the same succession of locations. By storing these segments in the database we will be able, after segmenting many routes from different users, to compute different

metrics for each segment based on the data from one user or all the users, using different time combinations. In Figure 6, the blue points are the location values received from the Directions API and the green ones are the locations got from the mobile application.

After requesting the segments from the API we had the entire route used decomposed into segments and the actual locations provided by our application, which each had its own timestamp. In order to calculate meta-data to characterize each segment, we needed to estimate the timestamps of each start and end location of each segment so we can then query the data from the sensors and determine the segment characteristics. To do this we attempted to find a segment determined by the locations from the mobile applications that is similar to the segment we are looking for, so we can approximate those segments have the same speed and then to determine their timestamps based on the timestamps of the points from the mobile application. First we calculated the bearing of the segment from the Directions API determined by a point with the same bearing, but by 5 meters from the start point and a point with the same bearing but right with 5 meters from the end point of the segment. Then from each one of the determined points we go left and right by 5 meters with a ninety degree bearing, creating a rectangle that contains in its center the original segment given by the Google Maps Directions API, like in Figure 7. A rectangle is used so we only take into account when approximating the time stamps the points that are actually part of that particular road. If two roads are really close to one another and the route is active on both roads we have to ensure that we are not using locations that will add time errors to the approximation.

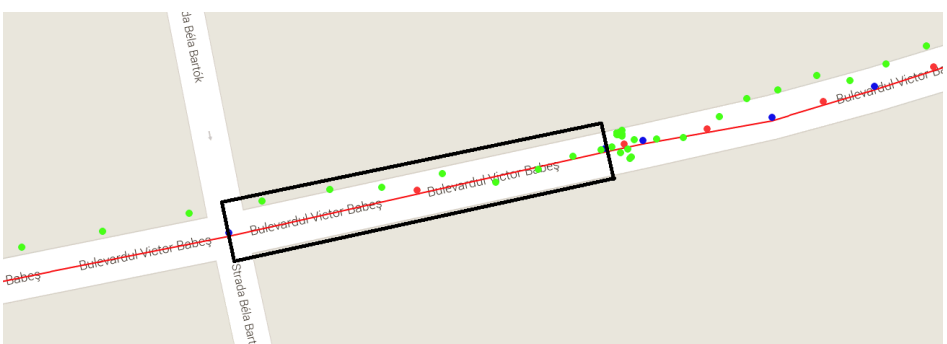


Figure 7: Segment mapping algorithm

After determining the rectangle for a segment, we determine which of the location points sampled by the sampling application are inside the rectangle and then determine the location that is closer to the segment start point and the location that is closer to the segment end point. Based on these two points, we then determine the speed on this particular segment and then, from the first point sampled by the application that is inside the rectangle we approximate the timestamp for the segment start and from the last point sampled by the application that is inside

the rectangle we approximate the end point.

Knowing the coordinates of the extremities for each segment that is part of the current route and also the timestamps for each one of them, we are making an AJAX request to the server where we transfer all the data mentioned above about each segment. The server checks whether the received segment is already stored in the database in the segments table and if it is not it is added. Then a query is assembled to get the data collected by the mobile application between the timestamps corresponding to the segment start and end point; and by using it we calculated the speed, the acceleration, the light intensity and the sound intensity for on the segment and we store it in the segment_data table. In this table we will have multiple entries that characterize the same segment and different periods of time and in order to have access to quick data about a particular segment without performed any intense calculations we calculate the metadata for the current segment and store it together with the segment data, as specified by the following attributes associated with path segments:

- The minimum, average and maximum speed
- The minimum, average and maximum acceleration
- The minimum, average and maximum light level
- The minimum, average and maximum sound level
- The number of unique users that have passed through the segment
- The number of passes through the segment, independent of the user

After the processing of the entire pool of segments for the current road is completed on the map, red segments will be drawn from each segment at the start location to each segment at the end location representing segments successfully characterized. A segment can be characterized only if in its rectangle there are at least two location points sampled from the mobile application, otherwise there will be no red polyline between the start point and end point of the segment, meaning it has not been characterized. In the center of the red polyline a pin will be displayed which will open an info window with the metadata that characterizes each segment. The info windows tell us how many users contributed to the segment characterization and the number of passes through the segment; and it is structured as a table containing minimum, average and maximum data about the speed, acceleration, light intensity and sound intensity calculated for the segment, like that shown in Figure 8

Additionally, on each location point sampled from the mobile application we included an info windows containing the point's timestamp and the speed at that particular point as retrieved from the GPS sensor, so the user can analyze the differences between the minimum, average and maximum speed calculated for the segment and the speed at a specific point, as shown in Figure 9.

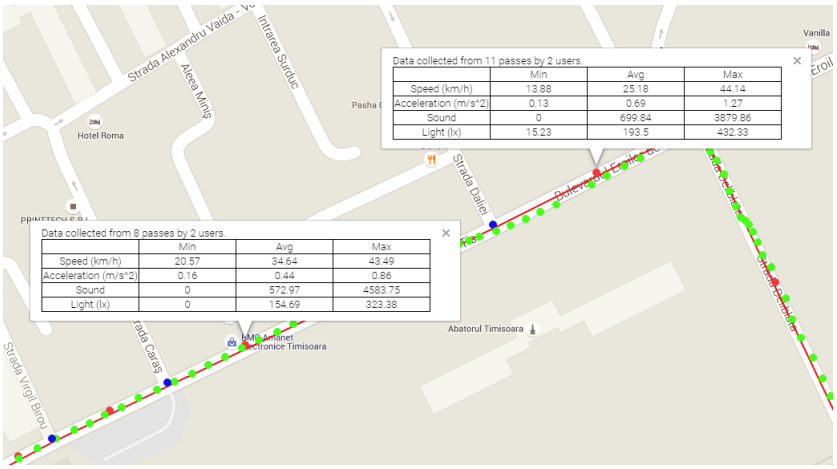


Figure 8: Visualization of the segment details

The speed variable used to calculate the minimum, average and maximum speed is computed from the segment start and end point timestamps, while the speed in the green points is provided by the GPS. The fact that these speeds are correlated demostrates that the approximation made for determining the timestamps for the segment start and end points did not introduce any significant errors.

4.3 Analysis Tools

Having got all the data about all the segments from the routes taken by the users stored in the database, we implemented a tool that displays a Google Map with all the segments processed, so the user can have an overview of the entire city. The tool presents several boxes from which the user can modify the data he wants to be displayed on the map. The following options that are available are listd in table 4.

Each segment on the map is color-coded, and it has one of four colors: green, light green, orange and red, green representing highest values and red representing lowest values as shown in Figure 10.

The available modes for the overview tool are listed in Table 5.

Along with the visual representation of different modes, the user can also analyze each segment by clicking on it; an info window will be displayed that displays data concerning the number of passes through the segment, the number of users collecting the data, the number of time intervals in which data was collected, the minimum, average and maximum speed, minimum, average and maximum acceleration, minimum, average and maximum light level and minimum, average and maximum sound level, as indicated in Figure 11.

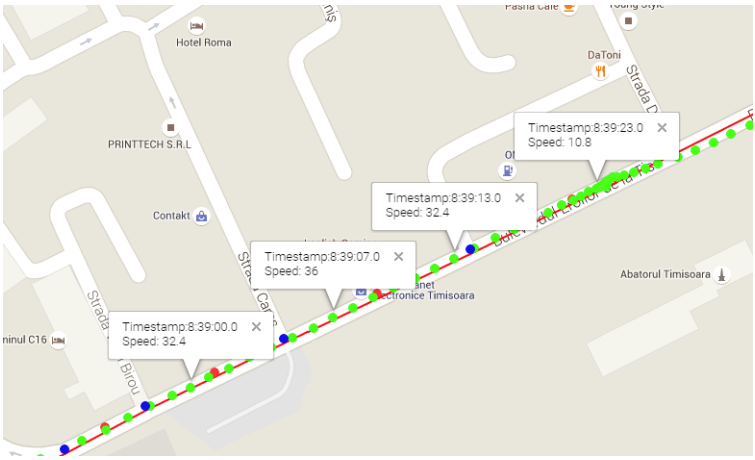


Figure 9: Visualization of the point details

5 Experimental Results

After finalizing the sampling application and testing it to ensure that all the data are sampled and stored correctly and after setting up the remote server we uploaded the sampling application in the Google Play repository so we could start collecting data from the daily routes that the users were taversing. The application was uploaded as a beta version, and only users that allowed us to collect data were allowed to use it. We present below some statistical data, to have an overview over the amount of data on which the research is based and also describing the three result components that can be determined by using the application:

- The user profiling,
- The area profiling,
- The best route estimation.

5.1 Statistical Data

The data that can be visualized and analyzed using the Web application and is presented here was sampled over six months, between the begging of January 2016 and end of June 2016 (6 months), but the process of data collecting is still in progress and only the processing of the routes has been temporarily put on hold to ensure data integrity and consistency.

So far the remote database has stored the following amount of data collected from user devices with the sampling application:

There were 364 routes recorded by the users in the application, routes which had the necessary data in order to characterize the resulting segments, and from the

Table 4: Analysis tool options.

Name	Description
Start hour	the data that was collected after a given hour will be displayed on the map
Stop hour	the data that was collected before a given hour will be displayed in the map
User	data collected by all the users or just the data collected by a particular user
Mode	specifies which property of the segment will be highlighted on the map

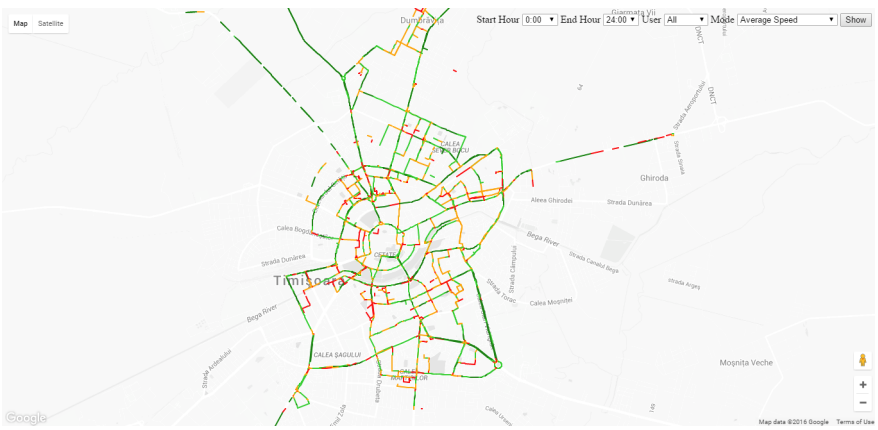


Figure 10: Aggregated data visualization

processing of those routes it resulted a total of 16,141 segments, most of them being in the Timisoara urban area. There are approximately 51.000 entries of segment data and 200,000 entries of segment metadata; in other words an average of four passes per segment. In this phase of the project 4 users who drive their own car, were involved in collecting daily traffic data. They installed the mobile client and let the application run for 6 months.

5.2 User Profiling

By using the overview tool in the Passes mode and selecting a certain user, we can see which routes and the areas that are most used by a given user, and which one determine his day-to-day travel route. In Figure 12, there are segments detailed in

Table 5: Overview of our analysis tool

Name	Description
Passes	The segments marked in green are those that have been used more extensively and the segments marked in orange and green are those with a lower passing count
Users	green segments reflect the fact that there is data was sampled from many users, while the segments marked in red and orange are those with data samples got from fewer users
Time intervals	green segments denote data sampled from many time intervals, while the orange and red segments have data sampled from less time intervals
Speed	green segments have a high minimum, average or maximum speed, while the red and orange segments have lower values for speed
Acceleration	green segments have a high minimum, average or maximum acceleration, while the red and orange segments have lower values for acceleration
Light level	green segments have a high minimum, average or maximum light level, while the red and orange segments have lower values for the light level
Sound level	green segments have a high minimum, average or maximum sound level, while the red and orange segments have lower values for the sound level

the Passes mode for two different users, depicting the routes used the most by each user. The green segments denote the routes most used by each user, while the red segments denote the routes that that the users do not use regularly.

Using the Passes mode for a specific user in combination with the timeline where we can select the start hour and the stop hour for the data that is being displayed on the map, we can determine the routes that the user used for traveling from home to work and from work to home by looking at the green segments. Between 07:00 and 10:00 we assume that the green route shown on the map matches the home-work route, while between 16:00 and 19:00 we can assume that the green route shown on the map matches the work-home route.

By using the Average speed mode with the username selected and within the desired time frame the user can estimate the best route to use to get from one point of the city to another by studying the green segments, as shown in Figure 13, which are the segments with the higher speed. In this way, the estimate is based only on the current user's data, based on his own driving style.

If no user is selected, then the estimate is based on all the user data and some users may have a more aggressive driving style than others, progressing this way

Figure 11: Aggregated data details visualization

with an increased speed. Hence, by selecting to just use the data collected by him, the use can estimate the best route for his own driving style.

As can be seen in Figure 13, for two different users the colors of the segment may differ for the same segment, influenced by the number of cars on the road at a certain time, but they are also influenced by the user's driving style, for certain segments the right user is more aggressive in traffic than the left one, telling us that it is important for the best route estimate to be made based on the user's own data, where possible.

The validation of the proposed paths was performed with user intervention using two metrics (path duration and path length) and user feedback. Both routes were taken by one user one after the other, and then a discussion took place about the two routes.

5.3 Area Profiling

By using the Passes mode with the data collected from all the users, also specifying the time frame for the analysis, we can determine which routes from a city or area are the most used one during the whole day or which routes from the city are the most used at certain times. The most used segments are also the most reliable as regards their metrics because they are calculated based on more passes through that route. The segments colored in green are the segments that were used more and there are multiple data entries in the database so the metrics are more accurate, while the red and orange segments have less passes through them, so the calculation

Table 6: Experimental database.

Sensor	Records count	Size [MB]
accelerometer	over 18.200.000	1.488
magnetometer	over 10.000.000	660
gyroscope	over 9.430.000	620
linear acceleration	over 5.500.000	360
light level	over 2.600.000	148
barometer	over 1.380.000	80
location (network/GPS)	over 404.000 entries	36
sound level	over 241.000	15
battery status	over 58.000	4
foreground application	over 50.000	4

of the metrics is not as accurate as the first one.

The reliability is also based on the number of users from which are getting the data from; the most used segments by different users can be examined via the Users mode. The green and orange segments tell us that through specific segments metadata are based on data collected from more users, while the red segments tell us that the metadata for that segment is calculated based on data collected from only one user.

By using the Average speed mode and using the data collected from all the users we can determine which segments are the most crowded in an area or city. Using the map and the color-coded segments, the user can identify which routes should he use and which ones to avoid at certain hours in order to reach his destination more quickly.

In Figure 14, the segments colored in green are the ones with speeds over 50 km/h, the ones colored in light green correspond to speeds over 35km/h, the ones colored in orange correspond to speeds over 20 km/h and the ones colored in red are corresponding to speeds under 20 km/h.

Besides using the Average speed mode to estimate the best route to get from one point on the map to another, the user can use other two approaches: an pessimistic one, by using the Minimum speed mode, where for each segment on the map the lower speed ever recorded for that particular segment is displayed and an optimistic approach using the Maximum speed mode, where the maximum speed ever recorded on the segment is displayed on the map. Below, in Figure 15 the two different options (optimistic and pessimistic) are illustrated to highlight the difference between them.

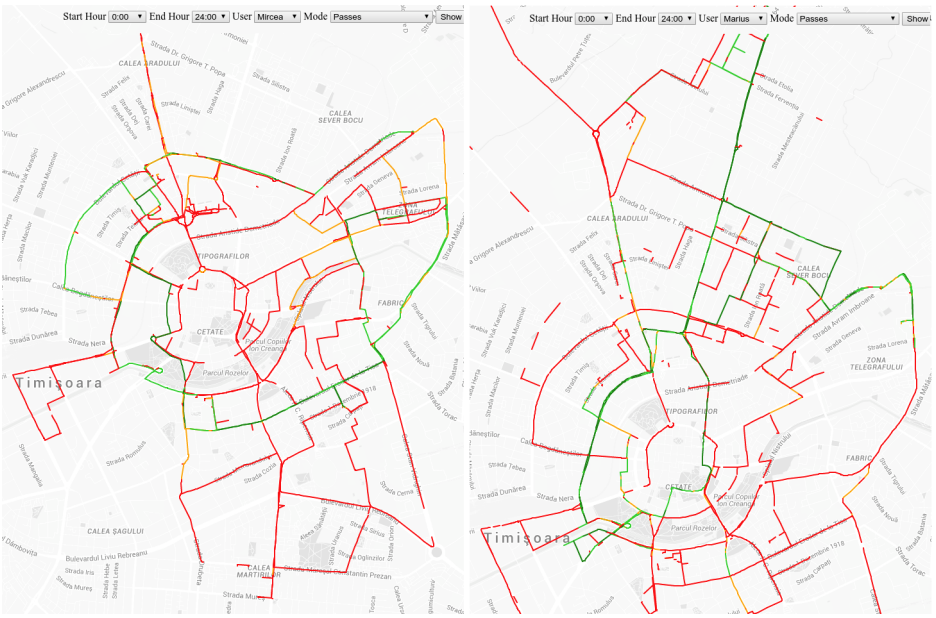


Figure 12: Visualization of the segments data availability

5.4 Best Route Estimation

By using the Average speed mode the user can study the possible routes between two points and then, based on the data gathered by himself or by other users, he can choose which ones from all possible routes he should choose and then calculate which one of these chosen is the best one in terms of distance or time, necessary to finish the route based on the average speed computed from the segment's historical data.

Assuming that the user wants to travel by car from point A to point B, as depicted in Figure 16, he can select the desired time frame and the segment data will be displayed on the map accordingly. He can also choose whether he wants the estimate to be based only on his sampled data, or whether he wants to use the data from all the users that have our application installed on their devices.

By analyzing the map the user can study some routes between the two points, but he can only choose some of them visually, by selecting the green roads for their greater speed. But having all the routes on the map composed of segments for which we have data collected, the user can choose and compare two actual routes, as in Figure 16.

Here, we called the two different routes between the point A and point B Route 1 and Route 2. By examining them visually on the map, the two chosen routes appear pretty similar, having almost the same distance and each one having more green segments than red and orange ones, meaning they are fast routes, see Figure 16.

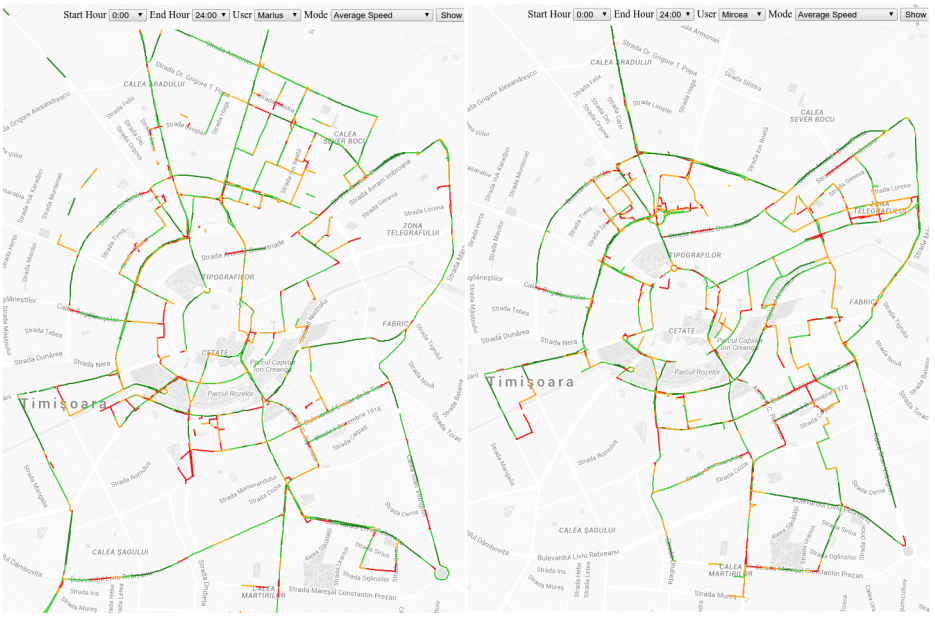


Figure 13: Average speed data visualization

By recomposing the routes segment by segment and having the length of each segment and the time needed to traverse it, we came to the conclusion that even through the route distances are not that different, Route 2 is much faster than Route 1:

- Route 1 is composed of 263 segments that total 6,675 meters and can be traveled on average in 19 minutes and 4 seconds with an average speed of 21 km/h.
- Route 2 is composed of 227 segments that total 6,276 meters and can be traveled on average in 11 minutes and 59 seconds with an average speed of 31.5 km/h.

Even though by looking at them on the map, the two routes look similar and they seem to have the same traveling speed, by calculating the sdtimates based on the aggregated data we find that the second route is thirty percent faster than the first one; by using the estimate and not just by looking at the map, the user can make a calculate choice based on this estimate.

The user can choose one of the three available modes to estimate the best route: the general mode, by using the Average speed mode, the pessimistic mode by using the Minimum speed mode and the optimistic mode by using the Maximum speed mode. The general estimation mode is depicted in Figure 19 and it is based on the average speed for each segment, computed from all the users that passed through

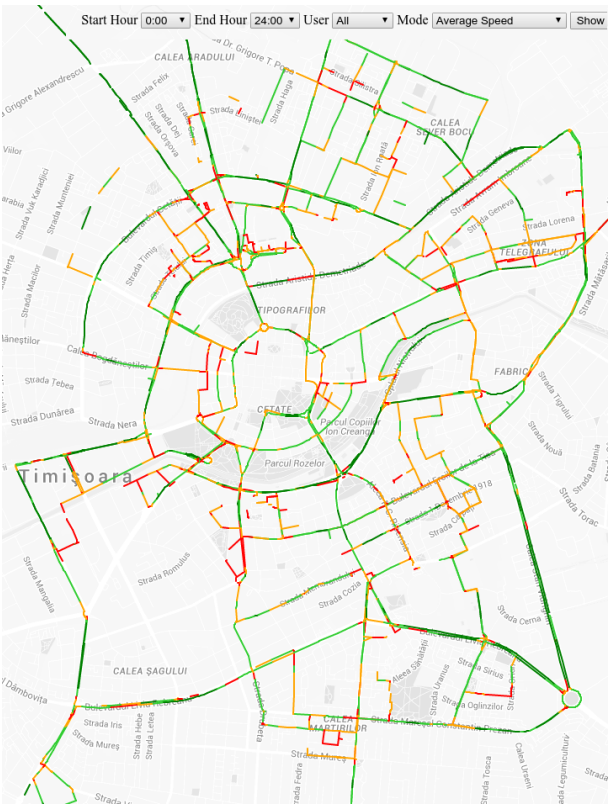


Figure 14: Visualization of the average speed data

that particular segment. The pessimistic estimation mode can be seen in Figure 20 and it is based on the lowest speed ever recorded for each particular segment. The optimistic estimation mode is depicted in Figure 21 and is based on the highest speed ever recorded for each particular segment.

We choose a route from one random point to another and we estimated the best route for each of the following cases: general, pessimistic and optimistic. For the route between the point A and point B, which is 2,594 kilometers long and is composed of 58 segments, the following times, were estimated:

- The general mode: 5 minutes and 20 seconds, with an average speed of 29 km/h
- The pessimistic mode: 28 minutes and 10 seconds, with an average speed of 5.5 km/h
- The optimistic mode: 4 minutes and 2 seconds, with an average speed of 38.5 km/h

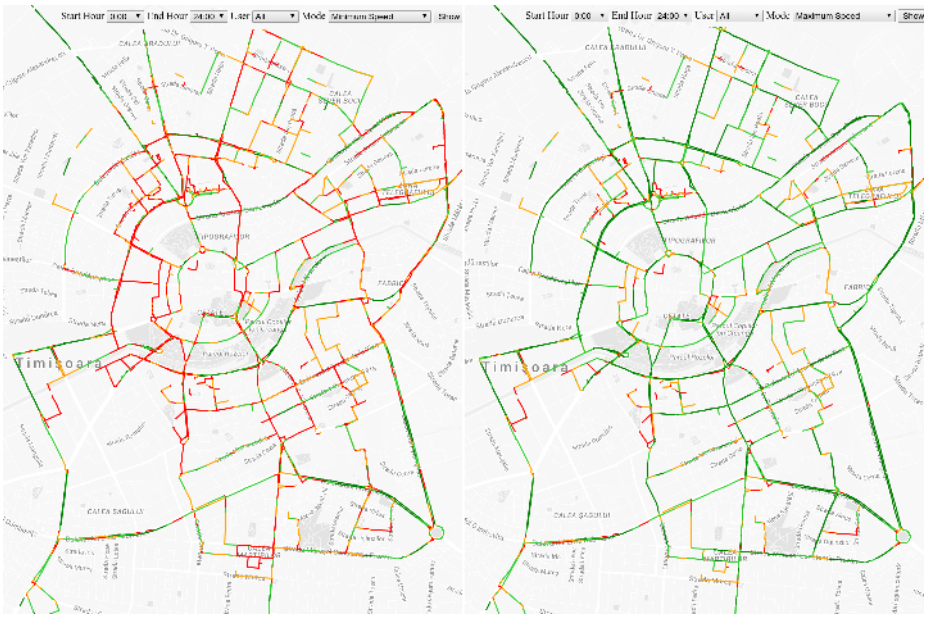


Figure 15: Optimistic and pessimistic data visualization

The difference between the pessimistic mode and the general mode is significant compared to the difference between the general mode and the optimistic mode, but this can be explained by the fact that the pessimistic mode takes into account all the events that ever slowed down the traffic for that particular route, while the optimistic one is determined by a regular, no-event traffic flow.

6 Conclusions

All the improvements that have been added to the mobile devices over the past few years, like the built-in sensors and the GPS, can be used in the user's favor. By collecting data concerning his day-to-day activities we can perform a user profiling and then provide personalized features for each user. This user profiling can be carried out by using non-sensitive data, like the data from the build-in sensors, but the main focus when collecting the specified data should be on the amount of energy that this operation is consuming. It is important to improve the energy efficiency of the data collection for the applications that provide certain features based on sensor data querying by dynamically modifying the sampling rates or enabling and disabling the sensors completely, depending in the context in which the user is at a certain time.

By sampling data from the accelerometer, gyroscope, magnetometer, barometer, linear acceleration sensor, light sensor and the locations of the network providers

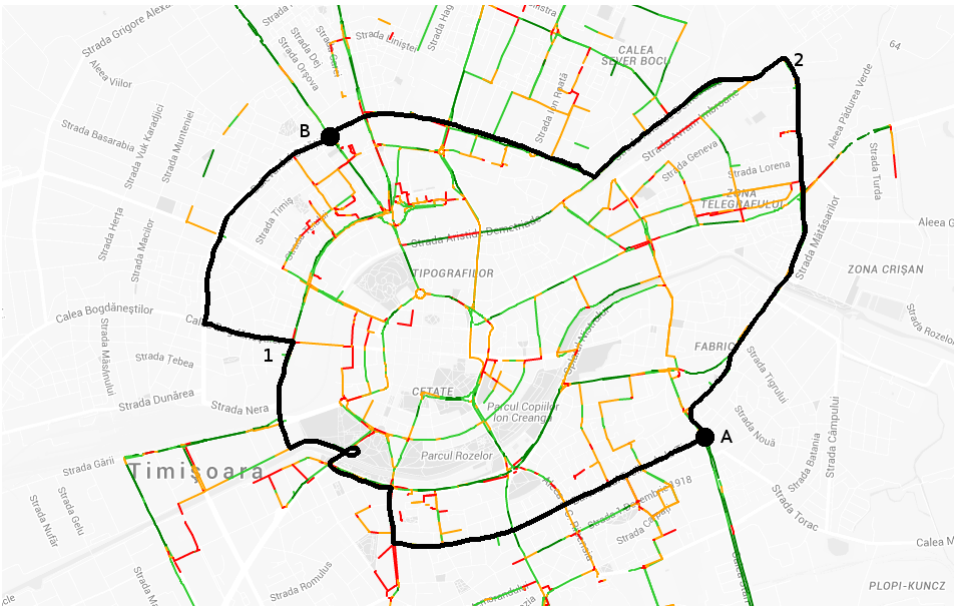


Figure 16: Comparative visualization of the routes

and the GPS while the user is driving, using public transport, using his bicycle or walking, we can compute all this data and characterize both the user and the routes he is using. The Google Maps Directions API allows us to query a certain route and it returns all the segments that make up that particular route, so after splitting it into segments we can compute the acceleration, speed, sound level, light level, distance and time frame metrics for each one of them. After gathering sufficient data from different users in different time frames traveling on different routes and after parsing all this routes, we can calculate metrics for each of the segments and then reassemble them to form different routes for which we can calculate the distance and estimate the average speed and required time to get from one point to another using either a general mode, based on the average speed of each segment, a pessimistic mode based on the minimum speed of each segment and an optimistic mode based on the maximum speed of each segment. The key aspect of the route estimation is that the user can choose the time frame in which he wants to travel and only the data sampled in that time frame will be taken into account to calculate the segment data. Another important point is that the user can select that the estimation should be based only on the data collected by him, this way ensuring that the estimate will be made according to his driving style.

Having 16,000 segments characterized and 51,000 segment data entries in the database means that we only have a small part of the Timisoara covered, so for the next time period our goal is to include more users in the collecting process and record and process as many routes as possible in order to characterize all the

segments from the metropolitan area of Timisoara and also gather data from all the existing hour-intervals, tso that the estimates will evermore reliable and accurate.

The applications created allow the users to gather their traffic data and map it to road segments in order to get the route best estimates in a pessimistic, optimistic or general way, for particular time intervals, based on the user profile or on a general profile with data gathered from all the users. The next phase in the application's development will be to implement an interface where the user can specify a starting point and a stopping point and as many intermediate points he finds necessary, to be able to define the routes he considers as his best options between the two points. The application will need the to process each route and estimate in a general, optimistic or pessimistic way which routes is better in terms of distance and transit time. Another key task will be the optimization of the remote database, because the size increases with each recorded route, which at this time is over 3.1 Gb. After processing the segments and calculating their metadata all the unnecessary data sampled from the sensors should be trimmed so that one can prevent huge amounts of raw sensor data from building up and being stored.

References

[1] Smith, B. L., Zhang, H., Mike, F., Green, M Final report of ITS Center project: Cellphone probes as an ATMS tool", University of Virginia: Smart Travel Lab, 2003.

[2] Wilkie, D., Jason S., and Ming L. Flow reconstruction for data-driven traffic animation. ACM Transactions on Graphics (TOG) vol. 32(4), pp. 89-99, 2013.

[3] Broring, A., Remke, A., Stasch, C., and Mollers, J. enviroCar: A Citizen Science Platform for Analyzing and Mapping CrowdSourced Car Sensor Data. Transactions in GIS, vol. 19(3), pp. 362-376, 2015.

[4] Tostes, A.J., Duarte-Figueiredo, F., Assuncao, R., Salles, J., and Loureiro, A. From data to knowledge: City-wide traffic flows analysis and prediction using Bing maps. Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing ACM, p. 12, 2013.

[5] Anwar, A., Till, N., and Ratti, C. Traffic origins: a simple visualization technique to support traffic incident analysis. Proceedings of IEEE Visualization Symposium (PacificVis), pp. 316-319, 2014.

[6] Zheng, X., Chen, W., Wang, P., Shen, D., Chen, S., Wang, X., Zhang, Q., and Yang, L. Big Data for Social Transportation IEEE Transactions on Intelligent Transportations Systems, vol. 17(3), March 2016.

[7] Goncalves, J., Goncalves, J.S.V., Rossetti, R.J.F., and Olaverri-Monrea, C. Smartphone Sensor Platform to Study Traffic Conditions and Assess Driving Performance Proceedings of IEEE 17th International Conference on Intelligent Transportation Systems (ITSC), October 8-11, 2014. Qingdao, China.

[8] Petrovska, N. and Stevanovic, A. Traffic Congestion Analysis Visualisation Tool Proceedings of IEEE 18th International Conference on Intelligent Transportation Systems, 2015.

[9] Dezani, H., Bassi, R., Marranghello, N., Gomes, L., Damiani, F., and Nunes da Silva, I. Optimizing urban traffic flow using Genetic Algorithm with Petri net analysis as fitness function. Neurocomputing, pp. 162-167, 2014.

[10] Motta, G., Sacco, D., Ma, T., You, L., Liu, K. Personal Mobility Service System in Urban Areas: the IRMA Project IEEE Symposium on Service-Oriented System Engineering, 2015.

[11] Gondim, H.W.A.S., do Nascimento, H.A.D., and Reilly, D. Visualizing Large Scale Vehicle Traffic Network Data - A Survey of the State-of-the-art International Conference on Information Visualization Theory and Applications (IVAPP), 2014.

[12] Xiong, G., Zhu, F., Dong, X., Fan, H., Hu, B., Kong, Q., Kang, W., Teng, T. A Kind of Novel ITS Based on Space-Air-Ground Big-Data IEEE Intelligent Transportation Systems Magazine, pp.10-23, Spring 2016

[13] Zhuo, W., Deng, R.H., Shen, J., Zhu, J., Ouyang, K. and Wu, Y. Multidimensional Context Awareness in Mobile Devices. MultiMedia Modeling, pp. 38-49. Springer International Publishing, 2015.

[14] Seungwoo, K, Lee, J., Jang, H., Lee, Y., Park, P., and Song, J. A scalable and energy-efficient context monitoring framework for mobile personal sensor networks. Mobile Computing, IEEE Transactions on 9, no. 5, pp. 686-702, 2010.

[15] Abdesslem, B., Fehmi, A.P., and Henderson, T. Less is more: energy-efficient mobile sensing with senseless. Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds. ACM, 2009.

[16] Wang, L., Zhang, D., and Xiong, H. effSense: energy-efficient and cost-effective data uploading in mobile crowdsensing. Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication. ACM, 2013.

Acta Cybernetica **23** (2017) 471–490.

The Connection between Antipatterns and Maintainability in Firefox

Dénes Bán[a]

Abstract

The notion that antipatterns have a detrimental effect on source code maintainability is widely accepted, but there is relatively little objective evidence to support it. We seek to investigate this issue by analyzing the connection between antipatterns and maintainability in an empirical study of Firefox, an open source browser application developed in C++.

After extracting antipattern instances and maintainability information from 45 revisions, we looked for correlations to uncover a connection between the two concepts. We found statistically significant negative values for both Pearson and Spearman correlations, most of which were under -0.65. These values suggest there are strong, inverse relationships, thereby supporting our initial assumption that the more antipatterns the source code contains, the harder it is to maintain.

Lastly, we combined these data into a table applicable for machine learning experiments, which we conducted using Weka [10] and several of its classifier algorithms. All five regression types we tried had correlation coefficients over 0.77 and used mostly negative weights for the antipattern predictors in the models we constructed.

In conclusion, we can say that this empirical study is another step towards objectively demonstrating that antipatterns have an adverse effect on software maintainability.

Keywords: static analysis, source code metrics, antipatterns, maintainability, correlation, machine learning

1 Introduction

In the area of source code analysis, there are many topics that are intensively studied. One of them is pattern recognition. Antipatterns are common solutions to frequently occurring problems which are supposed to incur decidedly negative consequences. However, even for the most widespread and universally accepted antipatterns, there is no substantial objective evidence that confirms their detrimental effects. To address this, we propose an empirical study intended to improve

[a]University of Szeged, E-mail: zealot@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.3

our understanding of the connection between antipatterns and source code maintainability.

As our subject systems, we selected 45 evenly distributed sample revisions taken from the `master` and `electrolysis` branches of Firefox between 2009 and 2010 – approximately one revision every two weeks. These revisions provided the basis for both antipattern detection and maintainability assessment. We extracted the occurrences of 9 different antipattern types and summed the number of matches by type. We also divided these sums by the total number of logical lines of the subject system for each revision to create new, system-level antipattern density predictor metrics.

Next, we computed corresponding maintainability values using a C++ specific quality model that calculates increasingly more abstract source code characteristics by performing a weighted aggregation of lower level metrics according to the ISO/IEC 25010 standard [14]. Its final result is a number between 0 and 1, which indicates the maintainability of the source code. Moreover, we adapted versions of the independent Maintainability Index [5] to get a secondary quality indicator.

With these data available, we attempted to answer the following two research questions:

- **RQ1: How does the number of antipatterns in a given system correlate with its maintainability?**
- **RQ2: Can the antipattern instances of a system be used to predict its maintainability?**

The paper is structured as follows. In Section 2, we list some related work, then in Section 3 we elaborate on our methodology. In Section 4, we discuss the results we obtained, then in Section 5 we overview some factors that might threaten the validity of these results. Lastly, in Section 6 we draw some pertinent conclusions and outline our plans for future work.

2 Related Work

Maintainability Trying to quantify complex software systems with a single maintainability index is not a new idea. Peercy [20] attempted to characterize subject systems using questionnaires as early as 1981. This, however, was a manual and mostly subjective effort.

Automatic source code analysis and metric extraction later led to metric-based maintainability models. One of the earlier – and more well-known – ones is the Maintainability Index metric (MI) published by Coleman et al. [5], which is a predefined formula that uses specific source code metrics to provide its result. Since it is still widely used to this day, we also included it in our investigations.

With the publication of the ISO/IEC 9126 framework [13], the expected structure and aspects of quality (and maintainability) models were more formally defined. It prescribes how to perform a weighted aggregation of objective, low-level

source code characteristics so it can obtain increasingly abstract values, thereby providing a high-level overview of the whole system. This aggregation is simply visualized by a graph whose leaf nodes are the source code metrics and the most abstract characteristic (in our case, the maintainability) is the root node. An example of this approach in practice is given by Antonellis et al. [2]. Similar to our approach, they also use expert opinion-based graph weighting, but they achieve it by using a technique called Analytical Hierarchical Processing. They conclude that this method helps domain experts to find connections between individual metrics and global maintainability as well as identify problematic areas.

Another example of the ISO/IEC 9126 framework in action is the probabilistic quality model published by Bakota et al. [3]. It also aggregates low-level metrics to arrive at the more abstract maintainability but instead of concrete "goodness values", it makes use of "goodness functions", and the leaf nodes of the dependency graph are treated as random variables. These goodness functions are built by analyzing a benchmark containing over 100 subject systems.

Our main approach uses the ISO/IEC 25010 [14] standard (a successor to ISO/IEC 9126) combined with the source code normalization method presented in [21]. While the model built there was meant for parallelization, not maintainability, the same principles apply. This notion is elaborated on in Section 3.3.

Antipatterns The two antipattern detection strategies closest to our own were published by Marinescu [18] and Moha et al. [19]. These studies both use source code metrics and threshold analysis and they both feature externally parametrized antipattern rules – but in these cases, the structure of the pattern is also customizable. Moreover, Moha et al. also utilize non-metric based, structural or even lexical cues which, although they cannot be so easily customized, have also been incorporated into our approach.

If preexisting benchmarks with known antipattern occurrences are available, machine learning becomes a viable option. Khomh et al. [15] built on the methodology of Moha et al. by making the decisions among parts of a complex ruleset more fuzzy with Bayesian networks. Another example was published by Maiga et al. [17], where they used Support Vector Machines to train models based on source code metrics to recognize antipattern instances. Here, however, we build machine learning models just to analyze the connection between the precomputed antipatterns and the maintainability of a given system.

In yet another approach, Stoianov and Şora [23] reduced pattern recognition to the resolution of logical predicates using Prolog. While this may seem radically different, there are similarities with our technique if we treat our metric thresholds and structural checks as the predicates and the programmatic source code traversal as Prolog's internal resolution process.

The Connections between Antipatterns and Maintainability As we mentioned earlier, little research has been done so far on finding an explicit connection between antipatterns and maintainability. One of these is our previous study [4],

where the two concepts were inversely related, while antipatterns were proportionately related to program faults (or bugs). Another is an investigation by Fontana and Maggioni [8] where they assume the connection and use antipatterns as well as source code metrics to evaluate software quality. Yet another is an empirical study by Yamashita and Moonen [25] where, after the refactoring of 4 Java systems, they conclude that antipatterns could provide experts and developers with more insights into maintainability than source code metrics or subjective judgment alone; however, a combined approach is suggested.

If we broaden our search from maintainability to include other concepts, antipatterns have been linked (among others) to:

- Comprehension by Abbes et al. [1], who concluded that, although single instances can be managed, multiple antipattern occurrences could have a significant impact and should be avoided,
- Class change- and fault-proneness by Khomh et al. [16], who concluded that classes participating in antipatterns are more change- and fault-prone, and
- Unit testing effort by Sabane et al. [22], who concluded that antipattern classes require substantially more test cases and should be tested with additional care.

On the other hand, if we just focus on maintainability, it has been positively linked to design patterns by Hegedűs et al. [11], refactorings by Szőke et al. [24], and version history metrics by Faragó et al. [6].

3 Methodology

The sequence of steps we took in order to answer our research questions is depicted in Figure 1.

The grey circles represent the different artifacts that exist between the steps of the process, while the white rectangles – which are explored in their own subsections – are the steps themselves.

3.1 Analysis

The analysis was conducted using a shell script that enumerated the 45 Firefox revisions, checked out the corresponding repositories (if not yet available) and updated them to the correct commit before initiating a build sequence. The core of the analysis was performed using the SourceMeter tool [7] developed at the Software Engineering Department of the University of Szeged.

It should be mentioned here that apart from the simple build script of `make -f client.mk`, our custom analysis configuration contained filters to skip the results of every command that matched the word "conftest" (a so-called hard filter) and to later skip any source code elements whose source code path information matched the filters described in Listing 1 (a so-called soft filter). These filters were obtained

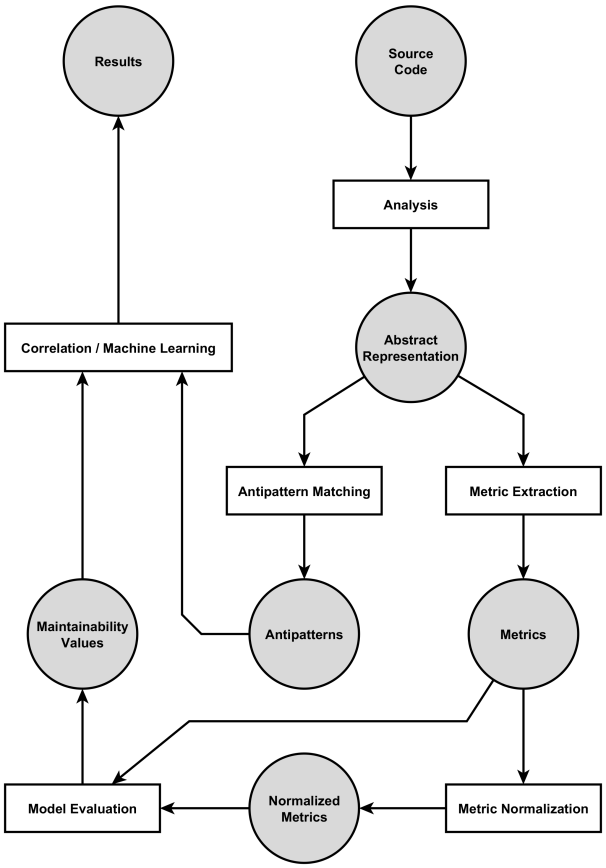


Figure 1: The methodology step sequence

via manual analysis of the Firefox repositories, pinpointing irrelevant or 3rd party code.

The lines in Listing 1 are applied in the order shown, allowing or disallowing a path based on the starting + or - character. So, for example, the first two lines mean that everything is filtered except for any content coming from the "repos" directory.

It should be added that the 45 Firefox revisions we selected are a subset of the Green Mining Dataset collected by Abram Hindle [12], as it is also our intention to relate antipatterns and software quality to energy and power consumption in the future.

Listing 1: The filter file for the analysis

```
-/
+/path/to/repos/
-/config/
-/testing/
-/build/
-/media/
-/security/
-/db/
-/jpeg/
-/modules/
+/path/to/repos/.*/modules/plugin/
+/path/to/repos/.*/modules/staticmod/
```

3.2 Metric Extraction

After performing our analysis, we extracted the metrics of the global namespace, which represent an aggregated, top-level view of the subject system. These metrics are the following:

- **HVOL** (**H**alstead **VOL**ume): if we let η_1 denote the number of distinct operators, η_2 the distinct operands, N_1 the total number of operators and N_2 the total number of operands, then $HVOL = N_1 + N_2 \cdot log_2(\eta_1 + \eta_2)$. From a C++ perspective, we will treat unary and binary operators (both arithmetic, increment, comparison, boolean, assignment, bitwise, shift and compound), keywords (e.g., return, sizeof, if, else, etc.), brackets, braces, parentheses, semicolons and pointer asterisks as operators, while the corresponding types, names, members, constants and literals will be treated as operands. Although this metric is usually used for single methods, it can be easily generalized to the system level.

- **TCBO** (**T**otal **C**oupling **B**etween **O**bjects): the CBO metric for a class means the number of different classes that are directly used by the class. Usage, among others, includes method calls, parameters, instantiations and attribute accesses as well as returnable and throwable types. **TCBO** is an aggregation of class-level CBOs to the system level, while **AvgCBO** (Average CBO) is defined as the ratio $TCBO/TNCL$.

- **TLCOM5** (**T**otal **L**ack of **CO**hesion in **M**ethods 5): for a class, LCOM5 measures the lack of cohesion, and it is interpreted as how many coherent classes the class could be split into. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between two nodes if and only if a common attribute or abstract method is used or a method invokes another. The value of the metric

is the number of connected components in the graph not counting those which contain only constructors, destructors, getters or setters. TLCOM5 is the sum of LCOM5s, while **AvgLCOM5** (Average LCOM5) is defined as the ratio $TLCOM5/TNCL$.

- **TRFC** (**T**otal **R**esponse set **F**or **C**lass): for a class, RFC is the number of local (i.e. not inherited) methods in the class plus the number of directly invoked other methods by its methods or attribute initializations. For the system, TRFC is the aggregated sum of RFCs, while **AvgRFC** (Average RFC) is defined as the ratio $TRFC/TNCL$.
- **TWMC** (**T**otal **W**eighted **M**ethods per **C**lass): the WMC metric for a class is the total of the McCC (McCabe's Cyclomatic Complexity) metrics of its local methods. For the system, TWMC is the sum of all WMCs, while **AvgWMC** (Average WMC) is defined as the ratio $TWMC/TNCL$.
- **TAD** (**T**otal **A**PI **D**ocumentation): the ratio of the number of documented public members of the system over the number of all of its public members.
- **TCD** (**T**otal **C**omment **D**ensity): the ratio of the comment lines of the system (TCLOC) over the sum of its comment (TCLOC) and logical lines of code (TLLOC).
- **TCLOC** (**T**otal **C**omment **L**ines **O**f **C**ode): the number of comment and documentation code lines in the system, where comment lines are lines that have either a block or a line comment, while a documentation comment line is a line that has (at least part of) a comment that is syntactically directly in front of a member. Note that a single line can be both a logical line *and* a comment line if it has both code and at least one comment.
- **TLLOC** (**T**otal **L**ogical **L**ines **O**f **C**ode): the number of code lines of the system, without the empty and purely comment lines.
- **TNA** (**T**otal **N**umber of **A**ttributes): the number of attributes in the system.
- **TNCL** (**T**otal **N**umber of **C**lasses): the number of classes in the system.
- **TNEN** (**T**otal **N**umber of **E**nums): the number of enums in the system.
- **TNIN** (**T**otal **N**umber of **I**nterfaces): the number of interfaces in the system. Note that although C++ lacks language support for the concept, we will treat classes with only pure virtual methods as interfaces.
- **TNM** (**T**otal **N**umber of **M**ethods): the number of methods in the system.
- **TNPKG** (**T**otal **N**umber of **P**ac**K**a**G**es): the number of namespaces in the system. Note that the word "package" here refers to a generalized object-oriented container concept which, in C++, directly maps to namespaces.

- **TNOS** (**T**otal **N**umber **O**f **S**tatements): the number of statements in the system.

It should be mentioned that the SourceMeter tool [7] did not have native support for some of the system-level metrics, including the Total and Average versions of CBO, WMC, LCOM5 and RFC, along with the aggregated Halstead Volume. The implementation of these computations was performed specifically for this study.

3.3 Metric Normalization

The metrics we have calculated so far may be viewed as complete from the perspective of the subject systems, but they cannot be related. They are, in a sense, absolute metric values and we have no way to tell, for instance, what an average WMC of 49.6 or a comment density of .31 *means* compared to each other. For this reason, it is desirable to normalize each metric value to the $[0, 1]$ interval using empirical cumulative distribution functions (or ECDFs). This method produces relative numeric values which indicate the ratio of how many of the available data points are smaller than a certain metric. These values are relative because they depend on the context they were evaluated in.

Let $(v_1, v_2, \ldots, v_n)$ be independent and identically distributed random variables with a common distribution function. The empirical distribution function is $\widehat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} I(v_i \leq x)$, where I is the indicator function; namely, $I(v_i \leq x) = 1$ if $v_i \leq x$ and 0 otherwise. For example, the empirical distribution function of variables 1, 1, 1, 1, 2, 2, 4, 4, 5, 5, 6, 6, 8, 9, 13, and 15 can be seen in Figure 2.

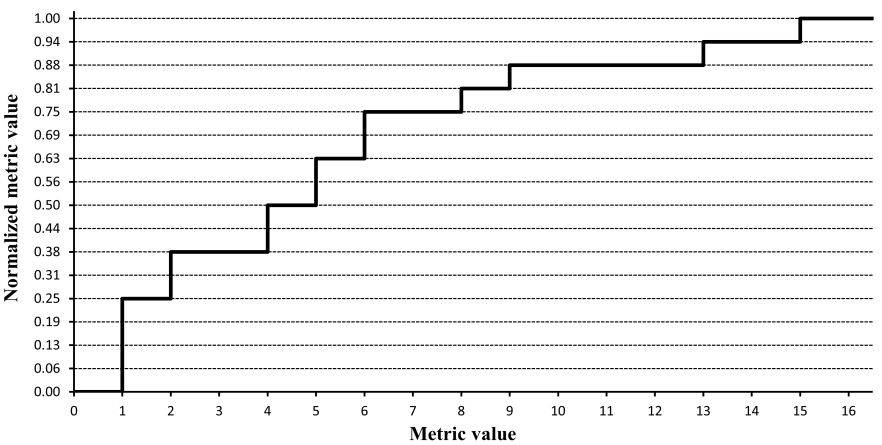


Figure 2: An example Empirical Cumulative Distribution Function [21]

Note that these normalized metrics will be greater for greater absolute inputs and smaller for smaller ones. However, the majority of our examined metrics are "the smaller the better", hence to facilitate a simpler mental model where values

closer to 1 mean *better* quality, we decided to invert this relation. The exceptions to this inversion – or, the "the bigger the better" metrics – are all documentation-related, namely TAD, TCD and TCLOC.

3.4 Antipatterns

For the automatic recognition of antipattern instances, we used our previously published static source code analyzer tool [4]. It currently recognizes the 9 types of antipatterns listed below. They are described in greater detail by Fowler and Beck [9], and here we will just provide a short informal definition and explain how we interpreted them in the context of our model.

Each antipattern implementation can define one or more externally configurable parameters, mostly used for easily adjustable metric thresholds. These are denoted by a starting **$** sign and their default values are listed in Table 1. Several of the object-oriented source code metrics referenced below coincide with the extracted metrics described in Section 3.2, while others are briefly explained in place.

- **Feature Envy** (FE): A class is said to be envious of another class if it is more concerned with the attributes of that other class than those of its own. It is interpreted as a method that accesses at least **$**MinAccess attributes, and at least **$**MinForeign% of those belong to another class.

- **Lazy Class** (LC): A lazy class is one that does not do "much", just delegates its requests to other connected classes – i.e., a non-complex class with numerous connections. It is interpreted as a class whose CBO metric is at least **$**MinCBO, but its WMC metric is no more than **$**MaxWMC.

- **Large Class Code** (LCC): Simply put, a class that is "too big" – i.e., it probably encapsulates not just one concept or it does too much. It is interpreted as a class whose LLOC metric is at least **$**MinLLOC.

- **Large Class Data** (LCD): A class that encapsulates too many attributes, some of which might be extracted – along with the methods that more closely correspond to them – into smaller classes and might be a part of the original class through aggregation or association. It is interpreted as a class whose NA metric is at least **$**MinNA.

- **Long Function** (LF): Similar to LCC, if a method is too long, it probably has parts that could (or should) be separated into their own logical entities, thereby making the whole system more comprehensible. It is interpreted as a method where the LLOC, NOS or McCC metric exceeds **$**MinLLOC, **$**MinNOS or **$**MinMcCC, respectively.

- **Long Parameter List** (LPL): The long parameter list is one of the most recognized and accepted "bad code smells" in code. It is interpreted as a function (or method) whose number of parameters is at least **$**MinParams.

- **Refused Bequest** (RB): If a class refuses to use its inherited members – especially if they are marked "protected," through which the parent states that descendants *should* most likely use it – then it is a sign that inheritance might not be the appropriate method of implementation reuse. It is interpreted as a class that inherits at least one protected member that is not accessed by any locally defined method or attribute.

- **Shotgun Surgery** (SHS): Following the "Locality of Change" principle, if a method needs to be modified then it should not cause a demand for many other – especially remote – modifications, otherwise one of those can easily be missed leading to bugs. It is interpreted as a method whose NII (Number of Incoming Invocations, i.e., the number of the different methods or attribute initializations where this method is called) metric is at least **$**MinNII.

- **Temporary Field** (TF): If an attribute only "makes sense" to a small percentage of the container class then it – and its closely related methods – should be decoupled. It is interpreted as an attribute that is only referenced by at most **$**RefMax% of the members of its container class.

It should be mentioned that in addition to these single antipatterns, we collected a "SUM" value, which is – not surprisingly – defined as the sum of all types of antipatterns in the given subject system. Also, we calculated densities for each absolute antipattern listed above, meaning that for every "AP" antipattern there is an "AP_DENS" metric available, computed as the ratio $AP/TLLOC$.

Table 1: Antipattern default thresholds

Antipattern	**Parameter**	**Value**
FE	MinAccess	5
FE	MinForeign%	80%
LC	MinCBO	5
LC	MaxWMC	10
LCC	MinLLOC	500
LCD	MinNA	30
LF	MinLLOC	80
LF	MinNOS	80
LF	MinMcCC	10
LPL	MinParams	7
SHS	MinNII	10
TF	RefMax%	10%

3.5 Maintainability Models

In order to assess the maintainability of the systems we analyzed, we created an expert opinion-based maintainability model according to the ISO/IEC 25010 standard [14]. The standard states that Maintainability should be the weighted aggregation of 5 subcharacteristics, these being Analysability, Modifiability, Testability, Modularity, and Reusability. The weights of how they are to be aggregated – and how they themselves are computed from source code metrics – were derived from the results of a poll.

First, the 10 experts – each of whom is an academic or industrial professional with at least 5 years of experience in software engineering – had to distribute 100 points among the source code metrics (listed in Section 3.2) for each subcharacteristic, to express how much they think that metric affects the given subcharacteristic. The results of this step are summarized in Table 2.

Table 2: The results of the subcharacteristic votes

ID	Analysability	Modifiability	Testability	Modularity	Reusability
HVOL	25	26.6	25.7	11	13
AvgCBO	22	29.6	24.5	43	33
AvgLCOM5	6.5	4.7	7.4	8.5	7.5
AvgRFC	1.5	3	3.3	15	7.5
AvgWMC	10	10	10.8	5.5	9
TAD	7.3	7	2.5	3.3	7.9
TCBO	1	1.3	3.6	0	1
TCD	4.1	1.5	1.5	0	2
TCLOC	0.3	0.5	0	0	4.5
TRFC	0	0.5	1	0.5	0.5
TWMC	1	1	0	0	0
TLLOC	14.5	9.2	8.5	0	2.4
TNA	0	0	0.2	0	0
TNCL	5	3	5	0	0
TNM	1.4	1.3	3.1	0	0
TNOS	0	0	1	0	0
TNPA	0	0	0	0	1.4
TNPCL	0	0	0	4.6	2.4
TNPEN	0	0	0	0.4	1.3
TNPIN	0	0	0	5.7	3.5
TNPKG	0.4	0.8	0	0	0.2
TNPM	0	0	1.9	2.5	2.9

Next, they had to distribute another 100 points among the subcharacteristics themselves, expressing how much each of them affects the overall Maintainability. The results of this step are summarized in Table 3.

Table 3: The results of the Maintainability votes

Subcharacteristic	**Maintainability**
Analysability	28.5
Modifiability	26.5
Testability	14.2
Modularity	17.1
Reusability	13.7

Given these weights – and later dividing by 100 – we were able to obtain system-level Maintainability values for each of the given subject revisions in the $[0, 1]$ interval.

In addition, as mentioned in Section 1, we computed the two "traditional" Maintainability Index metrics [5], interpreting them using our static source code metrics as:

$$MI = 171 - 5.2 \cdot ln(HVOL) - 0.23 \cdot TWMC - 16.2 \cdot ln(TLLOC)$$

and

$$MI_2 = 171 - 5.2 \cdot log_2(HVOL) - 0.23 \cdot TWMC - 16.2 \cdot log_2(TLLOC) + 50 \cdot \sin(\sqrt{2.4 \cdot TCD})$$

We also calculated their modified counterparts (MI^* and $MI_2{}^*$), where we changed the Total WMC values to their corresponding averages. We did so to scale each part of the sum to the same magnitude because complexity (WMC) is the only component not inside a logarithm or sine and the TWMC values dominated every other term of the formulas.

3.6 Correlations and Machine Learning

Correlation is a statistical relationship between two sets of data denoting the strength of their dependence. Its values can range from +1 (strong relationship) to -1 (strong inverse relationship). We tested both Pearson's (which expresses linear dependence) and Spearman's correlation (which is a Pearson's correlation performed on the rankings of the original data).

Regression analysis is another statistical approach for estimating the relationships among variables. It seeks to predict how the typical value of the dependent

variable changes when any one of the independent variables change. It achieves this by providing an estimate for the dependent variable from a continuous interval. Its most important performance measure is its correlation coefficient, expressing how well the predicted values follow the tendency of the real value of the dependent variable.

In our current case, the dependent variable was one of the maintainability metrics (our ISO/IEC 25010-based Maintainability or one of the previously mentioned MI versions), while the independent variables were the counts and densities of the different antipattern types.

In this empirical study, we evaluated each of the following regression types: linear regression, multilayer perceptron, reduced error pruning tree, M5P tree and sequential minimal optimization regression.

4 Results

After all these preliminaries, we are now ready to address our two research questions.

4.1 Correlation Results

To address **RQ1**, we decided to calculate the Pearson and Spearman correlations between each antipattern and maintainability measure pair, summarized in Table 4 and Table 5, respectively. Note that a single star suffix (*) means that the correlation is statistically significant at the .05 level, while a double star (**) means a significance at the .01 level. Also, to help in quickly parsing these tables, any cell where the correlation coefficient is either positive or non-significant was marked in a light gray background, and a darker gray when it is significantly positive (the worst case from our perspective).

As these tables clearly show, most antipattern-maintainability pairs have a strong, significant inverse connection. There are a few marked correlations, mainly for Modularity and Reusability, but even in these cases the non-significant values are still negative, while the positive values are non-significant and weak. We highlight the correlations between the SUM and SUM_DENS antipatterns and our final Maintainability measure as these represent most closely the overall effects of antipatterns on maintainability. The corresponding values are -.658 and -.692 for Pearson, and -.704 and -.678 for Spearman correlation, respectively. Thus, in response to the first research question we conclude – based on these empirical findings – that there is a strong, inverse relationship between the number of antipatterns in a system and its maintainability. This supports our initial assumption that the more antipatterns the source code contains, the harder it is to maintain.

Table 4: Pearson correlations between different antipatterns and maintainability measures

Antipattern	MI	MI_2	MI^*	MI_2^*	**Analysability**	**Modifiability**	**Testability**	**Modularity**	**Reusability**	**Maintainability**
FE	-.985**	-.985**	-.857**	-.796**	-.830**	-.738**	-.785**	-.141	-.311*	-.657**
FE_DENS	-.659**	-.659**	-.303*	-.219	-.548**	-.637**	-.679**	-.538**	-.570**	-.661**
LC	-.825**	-.825**	-.933**	-.968**	-.732**	-.502**	-.519**	.274	.023	-.371*
LC_DENS	-.749**	-.749**	-.886**	-.943**	-.666**	-.425**	-.435**	.327*	.075	-.297*
LCC	-.987**	-.987**	-.864**	-.822**	-.862**	-.758**	-.791**	-.133	-.326*	-.674**
LCC_DENS	-.670**	-.670**	-.296*	-.249	-.624**	-.700**	-.711**	-.563**	-.638**	-.722**
LCD	-.768**	-.768**	-.555**	-.438**	-.531**	-.554**	-.611**	-.290	-.314*	-.526**
LCD_DENS	-.662**	-.662**	-.424**	-.298*	-.422**	-.483**	-.541**	-.344*	-.325*	-.477**
LF	-.988**	-.988**	-.883**	-.830**	-.885**	-.782**	-.830**	-.174	-.372*	-.710**
LF_DENS	-.820**	-.820**	-.530**	-.455**	-.783**	-.818**	-.866**	-.566**	-.688**	-.835**
LPL	-.961**	-.961**	-.931**	-.872**	-.781**	-.643**	-.704**	.014	-.160	-.544**
LPL_DENS	-.864**	-.864**	-.741**	-.649**	-.652**	-.594**	-.673**	-.157	-.253	-.542**
RB	-.985**	-.985**	-.852**	-.788**	-.820**	-.736**	-.783**	-.165	-.328*	-.662**
RB_DENS	-.930**	-.930**	-.714**	-.634**	-.757**	-.734**	-.786**	-.317*	-.435**	-.695**
SHS	-.988**	-.988**	-.850**	-.778**	-.837**	-.767**	-.811**	-.212	-.375*	-.698**
SHS_DENS	-.889**	-.889**	-.634**	-.538**	-.745**	-.771**	-.815**	-.450**	-.548**	-.754**
SUM	-.982**	-.982**	-.869**	-.791**	-.814**	-.735**	-.785**	-.164	-.319*	-.658**
SUM_DENS	-.910**	-.910**	-.712**	-.606**	-.731**	-.729**	-.783**	-.342*	-.438**	-.692**
TF	-.955**	-.955**	-.835**	-.740**	-.777**	-.723**	-.771**	-.199	-.330*	-.651**
TF_DENS	-.882**	-.882**	-.711**	-.592**	-.691**	-.695**	-.747**	-.317*	-.397**	-.653**

Table 5: Spearman's correlations between different antipatterns and maintainability measures

Antipattern	MI	MI_2	MI^*	MI_2^*	**Analysability**	**Modifiability**	**Testability**	**Modularity**	**Reusability**	**Maintainability**
FE	-.985**	-.985**	-.853**	-.809**	-.852**	-.749**	-.806**	-.161	-.370*	-.652**
FE_DENS	-.535**	-.535**	-.257	-.258	-.440**	-.532**	-.561**	-.550**	-.595**	-.609**
LC	-.757**	-.757**	-.936**	-.920**	-.755**	-.538**	-.572**	.224	-.044	-.381**
LC_DENS	-.542**	-.542**	-.777**	-.854**	-.517**	-.276	-.307*	.372*	.123	-.133
LCC	-.985**	-.985**	-.873**	-.839**	-.871**	-.754**	-.811**	-.131	-.354*	-.651**
LCC_DENS	-.482**	-.482**	-.213	-.258	-.439**	-.543**	-.550**	-.590**	-.655**	-.636**
LCD	-.731**	-.731**	-.484**	-.445**	-.509**	-.551**	-.590**	-.303*	-.365*	-.528**
LCD_DENS	-.626**	-.626**	-.355*	-.344*	-.373*	-.431**	-.474**	-.299*	-.318*	-.428**
LF	-.991**	-.991**	-.849**	-.800**	-.902**	-.821**	-.866**	-.242	-.453**	-.728**
LF_DENS	-.874**	-.874**	-.622**	-.608**	-.824**	-.837**	-.876**	-.500**	-.671**	-.821**
LPL	-.952**	-.952**	-.926**	-.856**	-.851**	-.696**	-.750**	.019	-.219	-.560**
LPL_DENS	-.904**	-.904**	-.707**	-.670**	-.715**	-.654**	-.708**	-.184	-.338*	-.580**
RB	-.976**	-.976**	-.819**	-.793**	-.829**	-.734**	-.794**	-.167	-.375*	-.646**
RB_DENS	-.911**	-.911**	-.706**	-.728**	-.735**	-.674**	-.730**	-.219	-.396**	-.614**
SHS	-.985**	-.985**	-.820**	-.768**	-.884**	-.827**	-.871**	-.291	-.487**	-.747**
SHS_DENS	-.907**	-.907**	-.694**	-.698**	-.787**	-.773**	-.812**	-.370*	-.538**	-.726**
SUM	-.978**	-.978**	-.806**	-.754**	-.847**	-.786**	-.834**	-.250	-.444**	-.704**
SUM_DENS	-.895**	-.895**	-.674**	-.641**	-.732**	-.726**	-.768**	-.332*	-.476**	-.678**
TF	-.945**	-.945**	-.746**	-.704**	-.785**	-.746**	-.796**	-.277	-.445**	-.681**
TF_DENS	-.843**	-.843**	-.595**	-.543**	-.675**	-.704**	-.739**	-.378*	-.484**	-.665**

4.2 Machine Learning Results

To answer **RQ2**, we compiled ten tables applicable for machine learning experiments – one for each maintainability measure. These contained every antipattern type as predictors and the values for their chosen maintainability measures as targets for prediction. We then ran these tables through all five regression techniques mentioned in Section 3.6 to see how well they worked in practice. The resulting models were later tested with a 10-fold cross-validation, and the corresponding correlation coefficients are shown in Table 6.

Table 6: Correlation coefficients of the machine learning models

	Linear Reg.	**MLP**	**REPTree**	**M5P**	**SMO Reg.**
MI	.9991	.9969	.9079	.9983	.9993
MI*	.9825	.9968	.8695	.9727	.9971
MI2	.9991	.9969	.9635	.9983	.9993
MI2*	.9864	.9689	.9033	.9799	.9858
Analysability	.8210	.9085	.7632	.9097	.9151
Modifiability	.8082	.9223	.7286	.8138	.8348
Testability	.8637	.9547	.8564	.8874	.8903
Modularity	.9082	.8915	.7461	.7589	.8757
Reusability	.8247	.8927	.6777	.6222	.8455
Maintainability	.8513	.9318	.7619	.8179	.8556

The high values of these coefficients suggest an affirmative answer to our second research question: antipatterns *can* be valuable predictors for maintainability assessment. The models we built weight the antipattern predictors with mostly negative values, but there are numerous positive instances as well. Further analysis of the structure of the models in the case of the Maintainability target revealed that some antipatterns *consistently* appear with negative weights more often than others. Moreover, this ordering of importance largely coincides with the above correlation magnitudes.

4.3 Lessons Learned

The most obvious lesson learned, based on these results, is the measurable detrimental effect of antipatterns on maintainability. Moreover, the conclusion we drew from the correspondence between correlation values and negative model weights is that there could also be an *order of importance* among the antipatterns studied here.

The most important ones to avoid appear to be Long Functions, Large Class Codes and Shotgun Surgeries. The frequently suggested refactorings for the first

two antipatterns are "Extract Method" and "Extract Class", respectively. As for Shotgun Surgery, the main goal is to reduce coupling by moving or extracting methods or fields, or even identifying a common superclass.

Refused Bequests and Temporary Fields seem less dangerous. The former can be fixed with "Replace Inheritance with Delegation" or by extracting an even more abstract superclass to house just the common members, while the latter is often corrected with "Extract Class" – which can coincide with extracting a method object.

And finally, Long Parameter Lists, Feature Envies, Lazy Classes and Large Class Data instances can be more easily tolerated. However, these can also be eliminated using techniques given in [9]. Long Parameter Lists have "Preserve Whole Object" or "Introduce Parameter Object"; Feature Envy has "Move Method" or "Extract Method"; Lazy Classes may vanish if their functionality is inlined or their connections are introduced to each other without the middle man; and lastly, Large Class Data can be solved – again – with "Extract Class".

The key point of these observations is that developers should concern themselves more forcefully with the organization of source code, and not just its behavior, since the work they put in in advance seems to lead to an easier maintenance phase, while the performance overhead introduced by the extra classes and methods is negligible.

5 Threats to Validity

There are a few aspects that might possibly threaten the validity of our results. One is that the antipattern matches might not be correct. While finding antipattern instances is far from being a solved problem, we tried to acquire reliable statistics by implementing widely recognized antipatterns with usual/recommended threshold values and previously published tooling support.

Imprecise maintainability scores could also skew our results. To combat this, we decided to utilize static, independent source code metrics and expert opinion-based weight determination, all the while adhering to the guidelines of an international standard.

To ensure that the connections we uncover were not just coincidental, we only included statistically significant correlations in this study. The connections could also be attributed to the fact that both the maintainability scores and the antipattern instances are – at least partially – based on the same static source code metrics. Despite the overlap, there are important differences, because the two concepts do not rely on the same aggregation level of metrics (method/class or system level) and antipatterns heavily incorporate other structural cues as well. We would also argue that the results *could* be meaningful even if the base set of metrics were identical, given that the mapping of concepts to metrics is plausible.

Lastly, the generalizability of these findings could be largely affected by the number of subject systems analyzed. Although a benchmark made from 45 versions of such a huge and complex software system can hardly be regarded as small, we intend to include more revisions and different applications as well.

6 Conclusions and Future Work

In this study, we analyzed 45 revisions of Firefox and calculated static source code metrics for each of them. Using these metrics, specific threshold parameters and structural information, we matched 9 types of antipatterns and their respective densities in each revision. Also utilizing these metrics, we calculated maintainability values based on the ISO/IEC 25010 software quality framework. After correlating these two sets of data, we found statistically significant inverse relationships, which we consider another step towards objectively demonstrating that antipatterns have an adverse effect on software maintainability. Moreover, our machine learning experiments indicated that regression techniques can attain high precision in predicting maintainability from antipattern information alone, suggesting that antipatterns can be valuable besides – or even instead of – static source code metrics in software maintainability assessment.

A possible next step for this investigation might be to analyze more Firefox revisions or include other C++ subject systems in another empirical study. Also, our selected Firefox revisions have runtime, power consumption and energy efficiency measurements as part of the Green Mining Dataset [12], and this provides us with the chance to relate antipatterns or maintainability to those concepts, too. We plan to calculate maintainability values at the class level as well (instead of at the system level), thereby – hopefully – gaining a more fine-grained view of how antipattern and non-antipattern classes relate to maintainability.

References

[1] Abbes, Marwen, Khomh, Foutse, Gueheneuc, Yann-Gael, and Antoniol, Giuliano. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Antonellis, P, Antoniou, D, Kanellopoulos, Y, Makris, C, Theodoridis, E, Tjortjis, C, and Tsirakis, N. A data mining methodology for evaluating maintainability according to iso/iec-9126 software engineering–product quality standard. *Special Session on System Quality and Maintainability-SQM2007*, 2007.

[3] Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., and Gyimóthy, T. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243–252, 2011.

[4] Bán, Dénes and Ferenc, Rudolf. *Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability*, pages 337–352. Springer International Publishing, Cham, 2014.

[5] Coleman, D., Ash, D., Lowther, B., and Oman, P. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[6] Faragó, C., Hegedus, P., Ladányi, G., and Ferenc, R. Impact of version history metrics on maintainability. In *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*, pages 30–35, Nov 2015.

[7] Ferenc, Rudolf, Langó, László, Siket, István, Gyimóthy, Tibor, and Bakota, Tibor. Source meter sonar qube plug-in. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 77–82, Washington, DC, USA, 2014. IEEE Computer Society.

[8] Fontana, F. A. and Maggioni, S. Metrics and antipatterns for software quality evaluation. In *Software Engineering Workshop (SEW), 2011 34th IEEE*, pages 48–56, June 2011.

[9] Fowler, M. and Beck, K. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999.

[10] Hall, Mark, Frank, Eibe, Holmes, Geoffrey, Pfahringer, Bernhard, Reutemann, Peter, and Witten, Ian H. The WEKA Data Mining Software: An Update. In *SIGKDD Explorations*, volume 11, pages 10–18. ACM, June 2009.

[11] Hegedűs, Péter, Bán, Dénes, Ferenc, Rudolf, and Gyimóthy, Tibor. *Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability*, pages 138–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[12] Hindle, Abram. Green mining: A methodology of relating software change to power consumption. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 78–87. IEEE, 2012.

[13] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[14] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.

[15] Khomh, F., Vaucher, S., Guéhéneuc, Y. G., and Sahraoui, H. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314, Aug 2009.

[16] Khomh, Foutse, Penta, Massimiliano Di, Guéhéneuc, Yann-Gaël, and Antoniol, Giuliano. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[17] Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., and Aimeur, E. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475, Oct 2012.

[18] Marinescu, Radu. Detection strategies: Metrics-based rules for detecting design flaws. In *In Proc. IEEE International Conference on Software Maintenance*, 2004.

[19] Moha, N., Guéhéneuc, Y. G., Duchien, L., and Meur, A. F. Le. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan 2010.

[20] Peercy, D. E. A software maintainability evaluation methodology. *IEEE Transactions on Software Engineering*, SE-7(4):343–351, July 1981.

[21] Rudolf Ferenc et al. *REPARA deliverable D7.4: Maintainability models of heterogeneous programming models.* 2015.

[22] Sabane, A., Penta, M., Antoniol, G., and Guéhéneuc, Y. G. A study on the relation between antipatterns and the cost of class unit testing. In *Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR*, March 2013.

[23] Stoianov, A. and Şora, I. Detecting patterns and antipatterns in software using prolog rules. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 253–258, May 2010.

[24] Szőke, G., Nagy, C., Hegedűs, P., Ferenc, R., and Gyimóthy, T. Do automatic refactorings improve maintainability? an industrial case study. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 429–438, Sept 2015.

[25] Yamashita, Aiko and Moonen, Leon. Do code smells reflect important maintainability aspects? pages 306–315. IEEE, September 2012.

Acta Cybernetica **23** (2017) 491–501.

A Parallel Interval Arithmetic-based Reliable Computing Method on a GPU

Zsolt Bagóczki[a] and Balázs Bánhelyi[b]

Abstract

Video cards have now outgrown their purpose of being only a simple tool for graphic display. With their high speed video memories, lots of maths units and parallelism, they can be very powerful accessories for general purpose computing tasks. Our selected platform for testing is the CUDA (Compute Unified Device Architecture), which offers us direct access to the virtual instruction set of the video card, and we are able to run our computations on dedicated computing kernels. The CUDA development kit comes with a useful toolbox and a wide range of GPU-based function libraries. In this parallel environment, we implemented a reliable method based on the Branch-and-Bound algorithm. This algorithm will give us the opportunity to use node level (also called low-level or type 1) parallelization, since we do not modify the searching trajectories; nor do we modify the dimensions of the Branch-and-Bound tree [5]. For testing, we chose the circle covering problem. We then scaled the problem up to three dimensions, and ran tests with sphere covering problems as well.

Keywords: CUDA, parallel Branch-and-Bound, circle covering

1 Introduction

Finding platforms for parallel computations is not at all easy and straightforward, but depending on the task CUDA can be a great choice. With its computational power and easy implementation, it not only lowers the runtime of our applications, but speeds up the development process itself. The platform developed by NVIDIA provides a powerful tool for GPGPU (General-Purpose computing on Graphics Processing Units).

Two advantages of CUDA are that it provides direct access to the GPU's virtual instruction set and that our processes are computed on dedicated, computing kernels. CUDA is an API, meaning that it is not a new programming language one has to learn, but rather a tool that can be paired with different widely known

[a]University of Szeged, E-mail: Bagoczki.Zsolt@stud.u-szeged.hu
[b]University of Szeged, Institute of Informatics, E-mail: banhelyi@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.4

and frequently used programming languages, and it is implementable in several well-known development environments (e.g. Matlab, MS Visual Studio). When it comes to number representation, it is important to know that while every video card supports the integer number representation, the implementation of the floating point number representation was not realized until the appearance of DirectX 9. Double precision on video cards was not present until 2008, before the GT200 series of NVIDIA and before the HD3000 and HD4000 series of AMD. In the more recent GPU architectures there is a way to implement it, and many of the recent function libraries include the double precision version of the functions, just like that in the interval arithmetic function libraries by NVIDIA that we use. For efficiency purposes it is highly advisable to use vector-based containers in our application, since the GPU itself works with vectors too, and it is capable of performing operations simultaneously on the elements of the vectors.

The GPU's architecture, however, limits the usability of the card on various types of computing, since it is specialized on executing operations on multiple data flows at the same time. This design ensures the possibility of parallelization, so the processes based on this principle are easy to parallelize. That is, in these cases, the GPU can ensure computations many times faster than the CPU could. The GPU was our primary choice, since we have to perform the same process on all of the elements of a dataflow.

Our program uses the parallel version of the Branch-and-Bound method. During the iterations of the algorithm, intervals bounding the input problem or a part of the original problem will be divided into subintervals, and this process will be executed on multiple subintervals simultaneously. The power of parallelizing will be significant when handling a large number of input circles, and when the boundaries of the circles are near the boundaries of our intervals, because in such cases the division of the input problem has to be repeated very often, hence the number of subintervals will be large. The possibilities of parallelizing the algorithm will be discussed in Section 2.3.

2 The approximating method based on Branch-and-Bound and interval arithmetic

2.1 Reliable methods

In numerical computations, most of the time it is not enough to give a rounded up or rounded down result for our problems since we often need the most accurate solution possible, within a minimal error range, if possible, with a very accurate error estimation that can be used in further calculations. Lots of significant mathematical proofs require this kind of reliability, but this requirement can be extended to more realistic problems, such as different economic problems, where a miscalculation, or an incorrectly calculated error range can result in the loss of a great deal of money.

To solve these kinds of tasks, we will utilize so-called reliable numerical methods. To understand what we mean by a method being reliable, we need to be aware of the cause of these expectations, namely the error. It may originate from different problems, which might for instance be the inaccuracy of the input data, the inaccuracy of the formulas used; or, in our case, the inaccuracy of the numeric representation we use. These errors may accumulate throughout the whole computation process, and significantly distort the results. It is easy to see that because of this distortion, when it is necessary to have exact results, the results must be handled differently.

In short, we can call a method reliable if it gives us the upper and lower bounds of our results, with a guaranteed error estimation at the end of the process.

2.2 Naive interval arithmetic, and its usage

In numerical analysis due to the finite precision of computers and estimations, we can lose significant digits and this will leave us with inexact results and possibly huge errors. Because of this, it is reasonable to demand that a method should be able to give interval bounds of our computational results instead of a rounded up or rounded down result. It provides both a lower and upper bound to the results, hence instead of exact values, we have to compute with intervals. Whenever we encounter a source of error like rounding, our intervals will keep expanding, but the reliable method will give us an estimation of the error that we can include in our future computations in order to get increasingly accurate results [2, 1].

On the video cards, to be able to utilize the GPU in our computations as much as possible, we will have to define both intervals, and interval arithmetic operations directly. For this, we will use the `cuda_interval_rounded_arith.h` and `cuda_interval_lib.h` function libraries provided by NVIDIA, in which we find all the required interval operations [6].

However, using one-dimensional intervals on their own will not be sufficient for our two-dimensional test cases. Implementing these is not a hard task with the mentioned function libraries, and the way to create multidimensional intervals is fairly obvious.

The shapes used will be represented using two-dimensional intervals. The unit square to cover will be stored in a two-dimensional interval, namely $[[x1, y1], [x2, y2]]$, where $x = (0, 1)$ and $y = (0, 1)$. With these two points, the unit square can be extended in the plane. For the circles, a class is used to store the data structure. The origin of the circles is stored in one, two-dimensional interval with an (x, y) coordinate pair in the plane, where both x and y are intervals. Then we store the square of the length of the radius as a simple interval (r^2). For an illustration of this, see Figure 1.

The most significant operation that had to be implemented using intervals is the checking method itself, which is used to determine the covering of a subproblem – a slice of the unit square – with a given circle. We check if the distance of the points in the square from the origin of the circle are smaller than the radius of the circle itself. In the Euclidean plane, the distance between two points can be found

via the formula

$$distance(p_1, p_2) = \sqrt{((p_{2,1} - p_{1,1})^2 + (p_{2,2} - p_{1,2})^2)}, \tag{1}$$

where $p_1 = (p_{1,1}, p_{1,2})$ and $p_2 = (p_{2,1}, p_{2,2})$ are two-dimensional points in the Euclidean plane. It is easy to see that it is not efficient to take the root of the sum for each iteration, so the square of the radius of the circles will be stored in advance, and only the square of the distance will be computed without taking the square root each time. All we have to do now is to compare the squares of the two values with the following formula:

$$sup(distance^2) < inf(r^2), \tag{2}$$

where sup is the supremum value of the interval and inf is the infimum value.

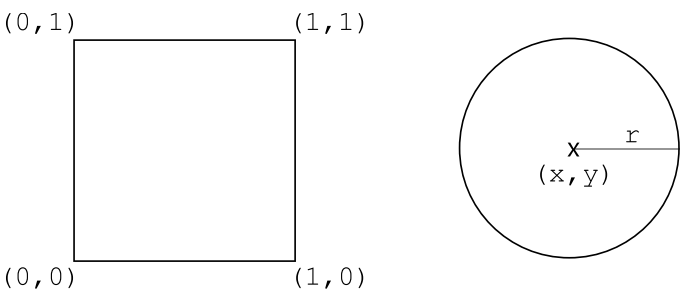


Figure 1: Two-dimensional shapes

2.3 The Branch-and-Bound algorithm

The implementation of our reliable method is based on a simplified Branch-and-Bound algorithm. The algorithm is well known and it is used widely in mathematical proofs, mathematical or computational optimization and in solving linear programming tasks.

The technique consists of two major substeps, namely the branching and the bounding. In the branching process, we divide our problem into two (dichotome branching) or more (polytome branching) subproblems, building an enumeration tree. Every division creates a new branch in this tree. The bounding process will give us the lower bound of the feasible solutions in the set of the solutions.

A simplified version of the algorithm is used with the goal to reliably prove any given quality of a two dimensional interval. A checking method is used to help in deciding whether the given quality is met. We discussed the checking method in Section 2.2 above. To improve the runtime, and to avoid infinite loops due to the infinite bisection of the subproblems when we bisect precisely on an interval boundary, we give a constraint for the length of the new subintervals. Let us call this constraint ϵ, the limit of the error. It should be mentioned that we cannot reliably

decide whether the quality is not actually met. This problem originates from the interval arithmetic we use; namely, there might be cases where the subintervals created by our bisections are already smaller than the given error limit ϵ, so a reliable proof only can be given if the given quality is actually met; otherwise it can not be proven whether the problem did not meet the quality or if in the bisection process, the result is already smaller than the given constraint.

The Branch-and-Bound algorithm is an easily parallelizable method [4]. We use node level parallelization, which means that we evaluate the nodes (which are the subproblems) simultaneously, and after all the evaluations are complete, we compare the results and continue the process based on the results [9]. This kind of parallelization is called low-level, or type 1 parallelization, since we will not modify the search trajectories or the dimensions of the Branch-and-Bound tree [7]. For this purpose, we shall use the computing cores of the CUDA supporting NVIDIA video cards, since they contain significantly more cores than a standard CPU does.

Algorithm 1 A parallel B&B algorithm.

Funct PIBB($param, \epsilon$)

1: Set the working buffer $\mathcal{B}_W := \{param\}$ and the result set $\mathcal{L} := \{\}$
2: Calculate current thread ID: tid
3: **while** $\mathcal{B}_W$ is not empty **do**
4: Pop v from $\mathcal{B}_W$ and split along all sides, and take the subproblem which belongs to our current thread: v_{tid}
5: **if** $maxWidth(v_{tid}) < \epsilon$ **then** *Termination rule*
6: $\mathcal{L}^{tid} = false$
7: **else if** $isMetQuality(v_{tid})$ is false **then**
8: Push v_{tid} into $\mathcal{B}_W$
9: **break**
10: **end if**
11: Synchronize threads
12: **if** $tid = 0$ **then**
13: **for** $i = 1, \ldots, sizeOf(\mathcal{L})$ **do**
14: **if** $\mathcal{L}^i$ = false **then**
15: **print** Cannot decide whether the quality is not met or the result is smaller than ϵ
16: **return** false
17: **end if**
18: **end for**
19: **end if**
20: **end while**
21: **print** The quality is met
22: **return** true

The correctness of the single core, non-parallel version of the algorithm has already been proven (see [3]).

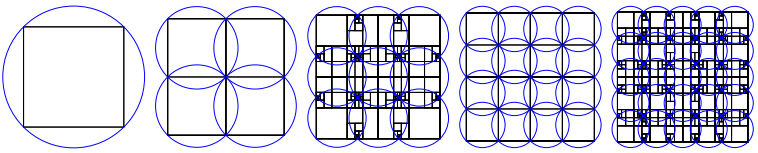


Figure 2: Test cases of circle covering

3 Test cases

3.1 Circle covering

The problem of circle covering [8] is the dual of the circle packing problem [12], where our goal is to give the densest covering of different shapes with n congruent circles. There are numerous proofs for both locally and globally optimal circle covering cases (see [11]).

For now, let us put aside the optimization problem for the covering cases. Our choice of testing was to place $n \times n$ circles on the Euclidean plane in such a way that they will certainly cover the unit square. The most obvious idea for this is to divide the diagonal of the square into n pieces, so the radius of a circle will be $r = \sqrt{\frac{2}{n}}$. In order to ensure the reliability of our computation, we need to add an error to the radius, which has to be bigger than our constraint ($\epsilon_r > \epsilon$). The reason is that near the tangential points of the circles, the distance should be 0 – which is smaller than our error range, causing the program to return with an undecidable result. During the testing phase, we used the values $\epsilon = 10^{-5}$ and $\epsilon_r = 10^{-4}$.

As we mentioned earlier, the method we use is a simple Branch-and-Bound based algorithm. The behaviour of it on the circle covering case is demonstrated by the first five test cases shown in Figure 2. We divide the sides of an interval into subintervals, and we check whether the given interval-piece is covered; then we continuously divide and check as explained above, until the result is positive or undecidable. The intervals will be divided up until the length of the intervals attain the length of the ϵ constraint, and below that point the method returns with an undecidable result.

When the number of the input circles is to the power of 2, the covering is unequivocal, meaning that it is decidable with a minimal amount of subproblems - in this case subboxes. For example, the $n = 1$, $n = 2$ and the $n = 4$ cases shown in Figure 2 illustrate the clarity and simplicity of the decision. The reason is that in these cases the dividing point of the intervals will be exactly on the intersections of the circles. Within a few divisions, we will reach a state where all the subproblems are the largest squares inside the circles, hence the circles will obviously cover the squares.

3.2 Expanding the circle covering problem to sphere covering

While solving circle covering problems in the Euclidean plane has a certain mathematical attraction, the world around us is three dimensional, hence we found it more useful to extend the problem to higher dimensions.

Firstly, we will have to rescale the input data. To store the spheres, we will use a class similar to the one we used to store circles, with a small modification to store the third dimension coordinates as well. The figure we wish to cover in this case is the unit cube, which is defined as the following: $[[x_1, y_1], [x_2, y_2], [x_3, y_3]]$, where $[x_i, y_i] = [0, 1]$, if $i = 1, 2, 3$. For data storage, we can use the stack buffer we have implemented earlier, which is accessible for both the CPU and GPU.

The n^3 spheres are placed along the space diagonal of the cube, and along lines parallel to this, which will ensure that they are tangential to each other. The radius in this case can be calculated from the 3D-space diagonal.

When rescaling the checking method, the distance between two points in a three-dimensional space is found using the usual distance metric in three dimensions. For a (p, q) pair of points, where $p = (p_1, p_2, p_3)$, and $q = (q_1, q_2, q_3)$, the following formula applies:

$$distance(p, q) = \sqrt{((q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2)}. \tag{3}$$

The covering is true when the distance between the points of an interval from the centre of the given sphere are less than the radius of the sphere (for which we store the squared value in the program). In this case, the subintervals will be cubes.

3.3 The test results

We ran the test cases first on 1, then on 2 CPU cores [10] as a benchmark for the GPU runtime testing, which was conducted on 256 CUDA cores. The runtime dependency on the number of circles can be clearly seen after a few cases, so we ended up running 20 test cases on all three of the core numbers.

We conducted the tests on the following hardware: AsRock Anniversary H97 motherboard, Intel Pentium G3450 CPU, 2*4GB DDR3 Kingston KVR16N11S8/4 memory, ASUS STRIX GTX950 2GB GPU and the Windows 10 Pro operating system.

The results of the runtime tests of the two-dimensional have been plotted in Figure 3. The results we got completely matched our expectations. By increasing the number of the circles, the runtimes also increased, and when we use more and more cores for the computations, the runtimes display an inversely proportional relation.

Our expectations for the three-dimensional problems were very similar to our expectations for the two dimensional one. The reason we still found it advisable to run the test cases in three dimensions is because of the increasing difference between the CPU and GPU runtimes that will provide an even better way of comparing them.

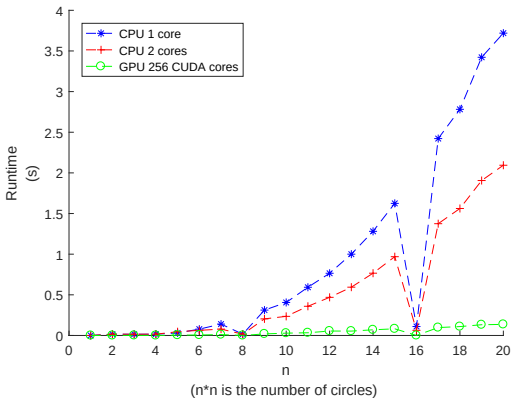


Figure 3: The runtime of the program on various test cases, with a different number of cores

The results of the tests can be seen in Figure 4. The values of the CPU and GPU runtimes do not display any irregularities; in fact they are just as we expected them to be; and the characteristics are almost identical to those we observed in the two dimensional test cases. The interesting part is when we compare the two results. Thanks to the parallelization on the GPU, the runtime of our program is never more than a few seconds, even when the process on the CPU terminates with a runtime of over an hour. This demonstrates the runtime differences much better than the few-minute differences measured in the two-dimensional testcases. Table 1 gives the actual measured runtime values in seconds.

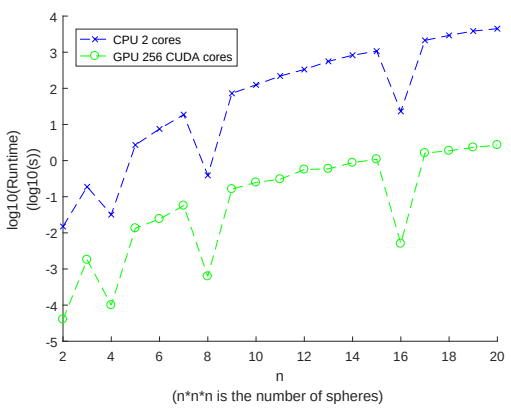


Figure 4: Test cases for sphere covering

As can be seen in both Figure 3 and Figure 4, there are certain exceptional cases. We mentioned previously that when the input number of circles or spheres is to the power of 2, the number of subboxes needed to be generated is minimal, which is the same as the number of the input shapes. In other words, the number of interval divisions made by the algorithm is low, therefore it should terminate rapidly and lead to significantly lower runtimes compared to those cases where the number of the input shapes was not to the power of 2.

Table 1: The results for the circle and the sphere covering test cases with different numbers of circles. The results are given in seconds.

n	Circle			Sphere	
	CPU (s)		GPU (s)	CPU (s)	GPU (s)
	1 core	2 cores	256 cores	2 cores	256 cores
1	0.000	0.000	0.000034	0.000	0.000035
2	0.016	0.016	0.000036	0.015	0.00004
3	0.015	0.015	0.001071	0.188	0.0018
4	0.016	0.016	0.000049	0.032	0.0001
5	0.031	0.046	0.004135	2.703	0.0134
6	0.078	0.063	0.005870	7.516	0.0242
7	0.140	0.078	0.010647	18.750	0.05675
8	0.016	0.016	0.000098	0.390	0.000627
9	0.312	0.204	0.019143	72.609	0.1624
10	0.406	0.235	0.028382	124.656	0.2489
11	0.593	0.359	0.033319	219.141	0.3049
12	0.765	0.469	0.054516	332.390	0.5652
13	1.000	0.594	0.053796	557.468	0.5919
14	1.281	0.766	0.069647	824.157	0.8735
15	1.625	0.969	0.081841	1069.800	1.0883
16	0.110	0.063	0.000297	23.094	0.00503
17	2.422	1.375	0.097214	2127.890	1.6199
18	2.782	1.562	0.109281	2915.480	1.8684
19	3.422	1.906	0.132101	3848.860	2.3156
20	3.719	2.094	0.136277	4481.250	2.7

4 Conclusions

During the testing of our reliable method we had a chance to try it out in a parallel, graphical computational environment. The test proved most informative, and the results once again met our expectations. It was a good illustration of where computing can not only be done fast and efficiently with supercomputers, but the average PC user can also have the opportunity and tools to experiment even if they are on a tight budget.

When handling processes or computations that are straightforward to parallelize, either at a tree or node level, it is recommended to exploit the power of the GPU, and if one does, one of the best options is CUDA. It provides a friendly, easily operable platform and supports most of the well-known programming languages like Java, C/C++, Python if not directly, then with the help of third party applications. The toolbox offers numerous process monitoring, examining opportunities, and ready made function libraries that contain algorithms and methods specially designed for GPU usage. Functions that are essential or indispensable in the daily work routine of a programmer or a mathematician are available.

A big advantage of CUDA and computing on GPUs overall is the accessibility of the video cards. Entry-level video cards are already able to solve parallel computing tasks and produce satisfactory results, and the 256 CUDA cores we used is roughly equal to the computational power of these cards. Cards supporting the latest version of CUDA are available in the price range of 100€, which makes them affordable even for regular PC users.

With the help of the readily scalable test cases, we got a good idea of the computing power of different architectures and their limitations. Now we can say from experience that when using approximation methods, a resource for computations like a video card is a very efficient tool, especially because it removes a big burden from our processor.

If we can reliably determine the properties for a circle, sphere, and hypersphere covering problem, then the next step is obvious: developing an algorithm which is able to give the optimal solution for the covering problems. In the future, we would like to develop a global optimization method based on our GPU-powered parallel method using interval arithmetic.

References

[1] Alefeld, G. and Herzberger, J. *Introduction to interval computations*. Academic Press Inc, New York, 1983.

[2] Alefeld, G. and Mayer, G. Interval analysis: theory and applications. *J. Comput Appl Math*, 121:421–464, 2000.

[3] Bánhelyi, Balázs, Csendes, Tibor, and Garay, Barna. A verified optimization technique to locate chaotic regions of Hénon systems. *Journal of Global Optimization*, 35:145–160, 2006.

[4] Casado, L.G., Martinez, J.A., Garcia, I., and Hendrix, E.M.T. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods Software*, 23:689–701, 2008.

[5] Clausen, Jens. Branch and bound algorithms - principles and examples, 1999.

[6] Collange, S., Daumas, M., and Defour, D. Interval arithmetic in CUDA. In *GPU Computing Gems Jade Edition*. 2012.

[7] Crainic, Teodor Gabriel, Cun, Bertrand Le, and Roucairol, Catherine. Chapter 1. parallel branch-and-bound algorithms, parallel combinatorial optimization, 2006.

[8] Friedman, E. Circles covering squares. `http://www2.stetson.edu/~efriedma/circovsqu/`, 2005.

[9] Garzón, E.M. and Garca, I. Parallel implementation for large and sparse eigenproblems. *Acta Cybernetica*, 15:137–149, 2001.

[10] Palatinus, E. and Bánhelyi, B. Circle covering and its applications for telecommunication networks. In *Proceedings of the 6th International Conference on Applied Informatics (ICAI2010)*. 2011.

[11] Sanders, Jason and Kandrot, Edward. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, United States, Ann Arbor, Michigan, 2010.

[12] Szabó, P.G., Markót, M.Cs., Csendes, T., Specht, E., Casado, L.G., and García, I. *New Approaches to Circle Packing in a Square – With Program Codes*. Springer, Berlin, 2007.

Acta Cybernetica **23** (2017) 503–512.

Storing the Quantum Fourier Operator in the QuIDD Data Structure*

Katalin Friedl[a] and László Kabódi[a]

Abstract

Quantum algorithms can be simulated using classical computers, but the typical time complexity of the simulation is exponential. There are some data structures which can speed up this simulation to make it possible to test these algorithms on classical computers using more than a few qubits. One of them is QuIDD by Viamontes et al., which is an extension of the Algebraic Decision Diagram.

In this paper, we examine the matrix of Fourier operator and its QuIDD representation. To utilize the structure of the operator we propose two orderings (reversed column variables and even-odd order), both resulting in smaller data structure than the standard one. After that, we propose a new method of storing the Fourier operator, using a weighted decision diagram that further reduces its size. It should be the topic of subsequent research whether the basic operations can be performed efficiently on this weighted structure.

Keywords: quantum algorithms, quantum Fourier operator, QuIDD

1 Introduction

Simulating quantum algorithms on a classical computer is a hard problem because of an exponential slow down. A quantum operator on n qubits can be represented on a classical computer by a $2^n \times 2^n$ matrix, so operations have exponential time and space complexity. In some cases, this can be decreased by having a good data structure. The Quantum Information Decision Diagram proposed by Viamontes, Markov, and Hayes [8] can store certain quantum operators in polynomial space, and compute some operations in polynomial time. They also have a quantum circuit simulator called QuIDDPro that employs their data structure [7]. Viamontes et al. [8] illustrated the use of the data structure on Grover's search [5]. In [4], using the different groupings of the operators, we gave an improved bound on the overall complexity of the simulation.

*This study is partially supported by the National Research, Development and Innovation Office (NKFIH), by the OTKA-108947 grant.

[a]Budapest University of Technology and Economics, Department of Computer Science and Information Theory E-mail: {friedl, kabodil}@cs.bme.hu

DOI: 10.14232/actacyb.23.2.2017.5

In this study, we focus on one of the most important quantum operators, namely the quantum Fourier operator. Using the structure of the operator, we present two natural reorderings of the columns that make the QuIDD representation of the operator smaller. Also, we propose a modification of the data structure to further reduce the number of nodes.

Although the motivation of this study is the efficient simulation of quantum algorithms, the problem itself is not limited to this area. The size of Ordered Binary Decision Diagrams (OBDDs) is an intensively researched topic. It has been proven that finding the optimal ordering for a Boolean function is NP-hard [3], but heuristics are useful, because in many areas (e.g. verification, model checking and computer aided design [9]) the size of the OBDD determines the performance of the applications. In some applications the width of the diagram is an important factor, but in this study we only concentrate on the number of nodes. It is known that the difference between the sizes for two orderings may be exponential, as in the case of the most significant bit of binary addition [2]. Our goal here is to analyze the efficiency of some orderings based on the structure of the operator.

Section 2 provides a description of the QuIDD data structure. After, Section 3 compares the number of nodes in the standard and modified orderings. Then Section 4 presents an idea of how to modify the data structure by weights to use significantly less nodes to store the operator. Lastly, in Section 5 we draw some pertinent conclusions and make a suggestion for future study.

2 The QuIDD data structure

The QuIDD data structure was specifically developed for quantum simulations [8]. For completeness, here we describe this data structure in details, based on [8, 4].

2.1 Binary Decision Diagram

For the representation of a Boolean function, the Binary Decision Diagram (BDD) is a popular tool [6]. It is a rooted directed acyclic graph, where every non-leaf node has exactly two child nodes. Each node is labelled with a variable of the function and the edges represent the 0 and 1 value of that variable. The value of the function is obtained by traversing the tree from its root following the edge representing the value of the variable. The value of a leaf is the value of the function. In the Ordered BDD the variables follow each other in a preset order. One of the problems of this representation is that it needs 2^n leaves and $2^n - 1$ internal nodes to store a function with n variables. The Reduced Order BDD (ROBDD) introduces reduction rules to achieve the following properties:

1. There are no nodes v and v' such that the subgraphs rooted at v and v' are isomorphic.

2. There is no internal node with both its edges pointing to the same node.

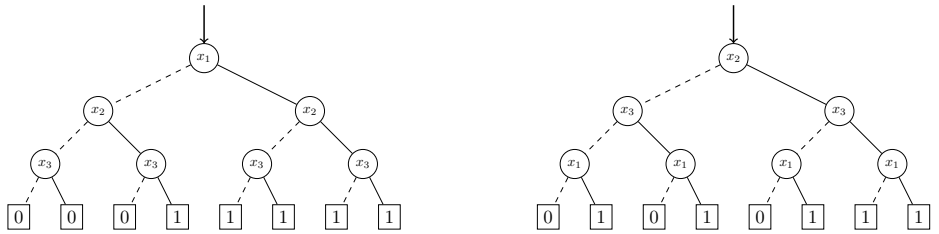


Figure 1: The BDD of function f using orderings x_1, x_2, x_3 and x_2, x_3, x_1. The solid lines represents the 1 edges, the dashed lines the 0 edges.

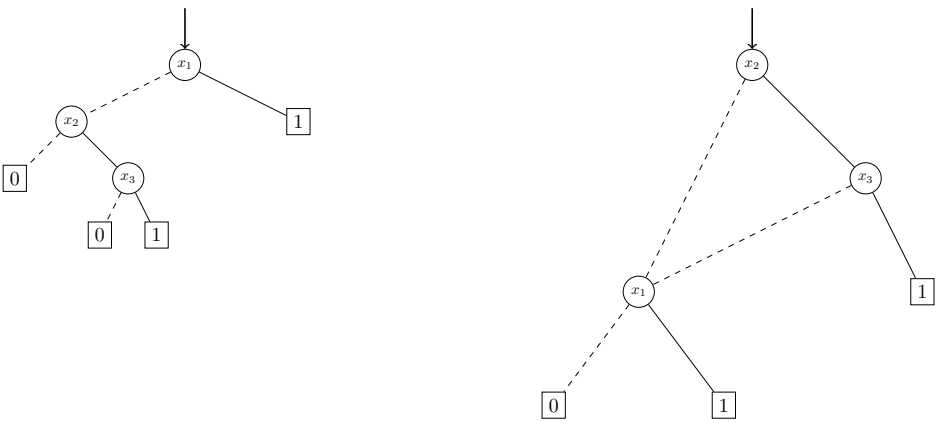


Figure 2: The ROBDD of the same function using the same orderings.

To illustrate the difference between the two structures, figures 1 and 2 show the BDD and ROBDD representations of the function

$$\begin{aligned} f(x_1, x_2, x_3) = &(x_1 \vee x_2 \vee \neg x_3) \wedge \\ &\wedge (x_1 \vee \neg x_2 \vee x_3) \wedge \\ &\wedge (x_1 \vee x_2 \vee x_3) \end{aligned}$$

with two variable orderings. As can be seen, the ROBDD can use significantly fewer nodes than the BDD and its size depends on the ordering.

Although a typical ROBDD is no longer a tree, the words root, leaf are still used for the corresponding nodes.

2.2 Quantum Information Decision Diagram

The QuIDD is a variant of the Algebraic Decision Diagram (ADD) [1], which is based on ROBDD. It is designed to store a matrix with complex elements. The leaves in the QuIDD are pointers to an array, which stores the values of nodes. These values may be real-valued or complex-valued.

When a matrix is stored, the variables of QuIDD correspond to the bits of the binary representation of rows and columns of the matrix. The variable R_i represents the i-th bit of the row numbers, and C_j represents the j-th bit of the column numbers. The numbering starts with 0, the 0th bit being the most significant one. The ordering of the variables is: rows and columns interleaved. This ordering is helpful if the matrices have some block structure, as usually happens when they are constructed from smaller matrices by tensor products.

The number of leaves is the same as the number of different values in the matrix. Let the *size* of a QuIDD be the number of its non-leaf nodes.

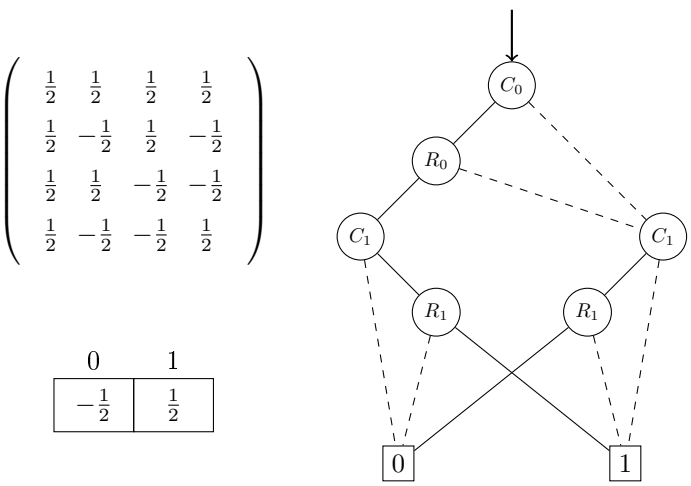


Figure 3: The Hadamard matrix acting on two qubits, and its QuIDD representation.

Example. On the diagram of figure 3, for the third (11) element of the second (10) row, one has to choose the 1 edge (solid line) at C_0, at R_0, and also at C_1, while the last step is the 0 edge at R_1. The 0 at the leaf leads to the value $-\frac{1}{2}$.

The 0th column of the matrix is constant so we can follow the dashed edge from C_0 to C_1 and to the leaf, where the 1 means that the value of the element is $\frac{1}{2}$. Notice that on this path there is no need for R_i nodes, because the value does not depend on the row.

In the following examples, for simplicity, a leaf contains the corresponding element of the matrix instead of a pointer to that element.

This data structure not only stores matrices, but computations can also be performed in this form. The basic operations of two matrices can be calculated in polynomial time and the result has a polynomial size. Namely, if there are two matrices with sizes a and b then their sum has size $O(ab)$, their product $O((ab)^2)$, and their tensor product $O(ab)$.

3 Storing the quantum Fourier operator using the QuIDD data structure

First, we examine the quantum Fourier operator using the standard QuIDD ordering. Unfortunately, in this way the recursive structure of the Fourier operator is not used. Then two other orderings are considered resulting in smaller sizes.

Proposition 1. *Without the reduction rules, the size of the Fourier operator is* $2^{2n} - 1$ *and it has* 2^n *leaves.*

3.1 Standard ordering

Recall that the standard ordering is $C_0, R_0, C_1, R_1, \ldots, C_{n-1}, R_{n-1}$.

Notice that every node in the structure corresponds to some contiguous submatrix formed by some neighboring rows and some neighboring columns of the matrix.

Proposition 2. *A node* C_i *describes a submatrix of size* $2^{n-i} \times 2^{n-i}$*. This submatrix can be obtained by fixing the first* i *bits of the row number and column number. A submatrix of node* R_i *is similar, but there the first* i *bits of the row number and the first* $i-1$ *bits of the column number are fixed. This submatrix has size* $2^{n-i+1} \times 2^{n-i}$*.*

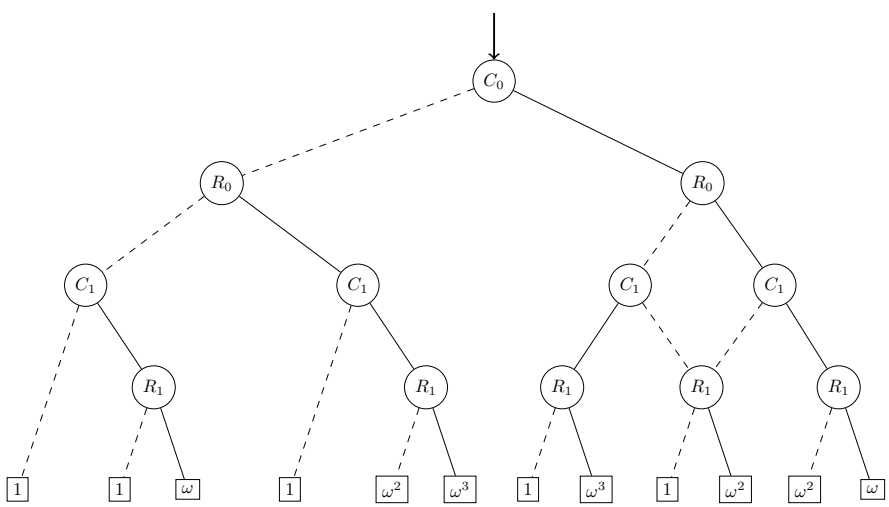


Figure 4: The Fourier operator of 2 qubits using the standard ordering.

Theorem 1. *Using the standard ordering, the size of the Fourier operator is* $\frac{5}{6} \cdot 2^{2n} - \frac{4}{3}$.

Proof. Let ω be a primitive 2^nth root of unity. In the Fourier transform the elements in the ith column are $\frac{1}{2^n} \cdot \omega^{k \cdot i}$, where k is the row number. When i is odd, all the different 2^n powers of ω appear. Since in neighboring rows the difference between the exponents in the same column is i, there are no two identical contiguous submatrices of size at least 2×2. This means that in QuIDD, identical nodes can appear only at the bottom. At the bottom, a node R_{n-1} corresponds to a 2×1 contiguous submatrix. From the above reasoning, these are different submatrices when the column index i is odd. Similarly, they are different when they come from different columns.

Let $i = 2^r \cdot a$ where $r \geq 1$ and a is odd. In this case the column is periodical with length of 2^{n-r}. Because an R_{n-1} node stores two elements, one column uses 2^{n-r-1} nodes.

For a fixed r there are 2^{n-r-1} possible a's. So at the bottom of the tree, the even columns use $\sum_{r=1}^{n-1} 2^{n-r-1} \cdot 2^{n-r-1} = \sum_{s=0}^{n-2} 4^s = \frac{4^{n-1}-1}{3} = \frac{1}{3}(\cdot 2^{2n-2} - 1)$ nodes.

The odd columns are all different, hence they use 2^{2n-2} nodes.

Since all 2×2 submatrices are different, the tree is complete up to the last level, this upper part having $2^{2n-1} - 1$ nodes. Adding it all up, we get $2^{2n-1} - 1 + \frac{1}{3}2^{2n-2} - \frac{1}{3} + 2^{2n-2} = \frac{5}{6}2^{2n} - \frac{4}{3}$. □

3.2 Different orderings

The study of Viamontes et al. [8] uses the standard ordering, but in general, the data structure works when all the matrices use the same ordering, which is not necessarily the standard one.

In the even numbered columns of the Fourier transform, the first 2^{n-1} rows are the same as the remaining ones. Moreover, they are the same as the Fourier operator of $n-1$ qubits. The following orderings are based on this observation.

Let us reverse the ordering of the column variables. In this ordering, we first check the last bit of the column, then the first bit of the row, and so on. Using the previous notation the ordering of the variables is $C_{n-1}, R_0, C_{n-2}, R_1, \ldots, C_0, R_{n-1}$. Figure 5 shows the structure for $n = 2$.

Theorem 2. *Using the reverse ordering of the column variables, the size of the quantum Fourier operator is* $\frac{2}{3} \cdot 2^{2n} - \frac{2}{3}$.

Proof. Let F_k denote the size of the Fourier operator of k qubits in the data structure. The subtree reached by the 0 edge from the root corresponds to the Fourier operator of $n-1$ qubits, and the subtree reached by the 1 edge from the root is a complete binary tree. Using this the recursion $F_n = 1 + F_{n-1} + 2^{2n-1} - 1$ is obtained. Unfolding this yields $F_n = 2^{2n-1} + 2^{2n-3} + \ldots + 2 = \frac{2^{2n+1}-2}{3}$; that is, $\frac{2}{3} \cdot 2^{2n} - \frac{2}{3}$.

□

Notice that this is approximately $\frac{4}{5}$ of the size of the standard ordering.

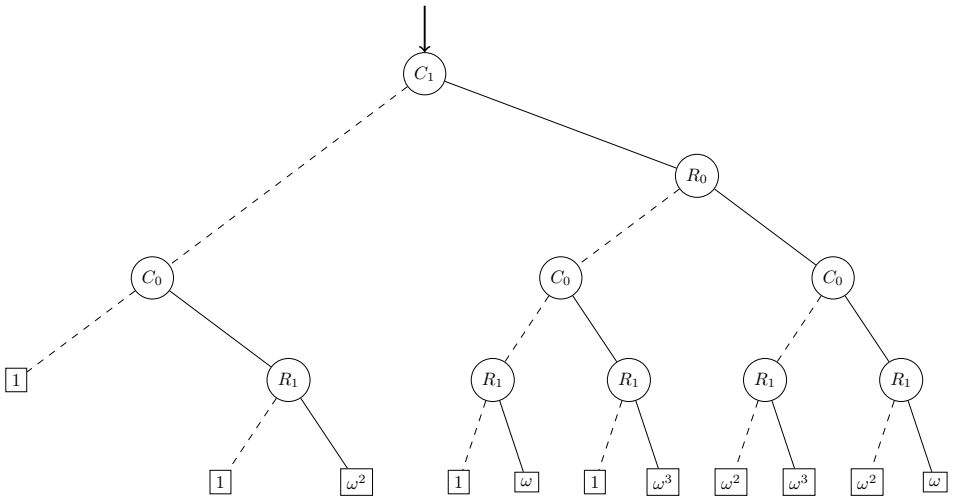


Figure 5: The Fourier operator of 2 qubits, with the column ordering reversed.

Without modifying the software of QuIDDPro, this reordering can be realized if instead of the Fourier operator F one uses the modified $F' = P \cdot F$ matrix, where P is the permutation matrix which reverses the binary form of the number of the columns. This is the same as the reversed QuIDD representation of the Fourier operator. However, there is a problem with this, namely the size of the permutation matrix P is large with this ordering.

There is another approach where we can find a good permutation matrix of small size. We know that the even numbered columns can be used to reduce the size of the QuIDD representation, so we need to collect them in the first half of the matrix. One solution to this problem is a permutation matrix P', which changes the order of columns to $0, 2, 4, \ldots, 2^n - 2, 1, 3, \ldots, 2^n - 1$. This way, we do not get the Fourier operator of $n-1$ qubits as a submatrix, but we still can exploit the fact that the 0 side of the first variable in the QuIDD representation is one level smaller, for symmetry reasons. Notice that for the Fourier operator of 2 qubits, this representation and the previous one are the same.

Intuitively, this means that we only have to store 3 of the $2^{n-1} \times 2^{n-1}$ submatrices, so the number of nodes will be around $\frac{3}{4} \cdot 2^{2n}$.

Theorem 3. *The size of the Fourier operator using this even-odd ordering is* $\frac{17}{24} \cdot 2^{2n} - \frac{4}{3}$

Proof. The two subtrees of the root behaves differently. The 1 side is a complete binary tree with $2n-1$ levels. Because it contains the odd columns of the operator, it cannot be compressed, and it uses $2^{2n-1} - 1$ nodes.

The 0 side contains the even columns. Here, only the $2^{n-1} \times 2^{n-1}$ submatrix has to be stored, because the upper and lower halves of the $2^n \times 2^{n-1}$ submatrix are the same. By Theorem 1, this subtree uses $\frac{5}{6}2^{2n-2} - \frac{4}{3}$ nodes.

Adding up the subtrees plus the root node, we get the desired $\frac{17}{24} \cdot 2^{2n} - \frac{4}{3}$. □

This representation is somewhat larger than the previous one, but it is still smaller than the size of the standard ordering, and the permutation matrix P' that is used to transform F can be stored efficiently.

4 Modifying the QuIDD data structure

Another way of exploiting the inner structure of the Fourier operator is to change the data structure slightly. Let us allow weights on the edges. Let the value of an element be the product of the weights on the path to the leaf and the element in the leaf. Using this model, if two subgraphs differ only by a constant multiplier, it is enough if we store only one of them.

This method can be combined, for example, with the reversed column ordering.

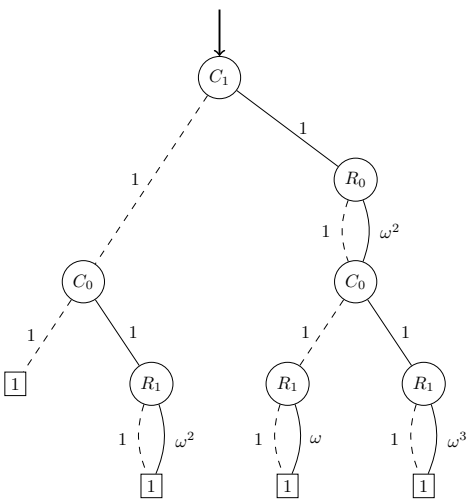


Figure 6: The Fourier operator of 2 qubits, column ordering reversed, using the weighted diagram.

Proposition 3. *The size of the Fourier operator using the weighted diagram and the reversed column ordering is* $3 \cdot 2^n - n$.

Proof. A recursion to calculate the number of the nodes is $F_n = F_{n-1} + 1 + 2^n - 1 + 2^{n-1} - 1$, because the 0 side of the root is F_{n-1}, and the 1 side consists of two complete binary trees – one for the rows, and one for the columns. This means that $F_n = 3 \cdot 2^n - n$. □

In this representation, it is sufficient to have only one leaf because the different values can be stored in the edge leading to the leaf.

Unfortunately, the operations described in [8] do not work on these diagrams. It would be interesting to investigate whether one could modify these operations so that they work on the weighted model.

5 Conclusions

While finding the optimal ordering of an ordered decision diagram is NP-hard, one can try to use the inherent structure of the matrix to improve the standard representation. The Fourier operator has an inner structure that can be seen after reordering its columns, and this reordering in the QuIDD data structure reduced the size by about 20%. A different, but more readily applicable reordering gave a reduction of about 15%.

An even bigger decrease in the size of the diagram came from allowing weighted edges. This change reduces the size of the data structure from $\Theta(2^{2n})$ to $\Theta(2^n)$ in the case of an Fourier operator of n qubits. The standard operations of [8] do not work on this diagram, so future research is necessary to decide if this weighted version can be directly used in simulations of quantum algorithms.

References

[1] Bahar, R Iris, Frohm, Erica A, Gaona, Charles M, Hachtel, Gary D, Macii, Enrico, Pardo, Abelardo, and Somenzi, Fabio. Algebric Decision Diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[2] Bollig, Beate. On the minimization of (complete) Ordered Binary Decision Diagrams. *Theory of Computing Systems*, pages 1–28, 2014.

[3] Bollig, Beate and Wegener, Ingo. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[4] Friedl, Katalin and Kabódi, László. An idea to improve QuIDD based quantum simulations. *Periodica Polytechnica. Electrical Engineering and Computer Science*, 59(2):48, 2015.

[5] Grover, Lov K. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 212–219. ACM, 1996.

[6] Jukna, Stasys. *Boolean Function Complexity: Advances and Frontiers*, volume 27. Springer Science & Business Media, 2012.

[7] Viamontes, George F., Markov, Igor L., and Hayes, John P. QuIDDPro: High-performance quantum circuit simulation. http://vlsicad.eecs.umich.edu/Quantum/qp/.

[8] Viamontes, George F, Markov, Igor L, and Hayes, John P. Improving gate-level simulation of quantum circuits. *Quantum Information Processing*, 2(5):347–380, 2003.

[9] Wegener, Ingo. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM, 2000.

Acta Cybernetica **23** (2017) 513–536.

A Clinical System Integration Methodology for Bio-Sensory Technology with Cloud Architecture

Abel Garai[a], Istvan Pentek[a], Attila Adamko[a], and Agnes Nemeth[b]

Abstract

Here, we outline the design, implementation, testing and evaluation phases of our bi-directional semantic and syntactic interoperability framework interconnecting traditional healthcare, industrial telemedicine and IoT wearable eHealth-domains. Specifically, our study demonstrates system interoperability among a hospital information system, an industrial telemedicine instrument and an eHealth smart wearable consumer electronic product through the Open Telemedicine Interoperability Hub (OTI-Hub) embedded in a hybrid Cloud architecture. The novelty of this study is the handling of Internet-of-Things smart healthcare devices and traditional healthcare devices through the same Cloud-based solution. This healthcare interoperability solution, service architecture and corresponding software engineering technique bridges technology barriers among the above-mentioned healthcare segments. Standard interoperability solutions exist and have already been described in related literature, but they are not applicable to the IoT healthcare devices and vice versa. Our study goes beyond isolated, individual interoperability solutions and seeks to bridge all major healthcare architecture frameworks including classical, telemedicine and eHealth IoT applications and appliances. This study presents the results of a two-year OTI-Hub Research Program. These experiments are manifestations of a trilateral cooperation among the University of Debrecen, Faculty of Informatics, the Semmelweis University 2nd Department of Paediatrics Pulmonology Division and an international hospital information system service provider.

Keywords: telemedicine, interoperability, bio-sensory monitoring, hybrid cloud, Hospital Information System, Health Level Seven, spirometry, Internet of Things

[a]Department of Information Technology, Faculty of Informatics, University of Debrecen, Kassai út 26, H-4028 Debrecen, Hungary, E-mail: {garai.abel, pentek.istvan, adamko.attila}@inf.unideb.hu

[b]3 Department of Pulmonology, 2nd Paediatrics Clinic, Semmelweis University, Tuzolto u. 7-9, H-1094 Budapest, Hungary E-mail: nemeth.agnes1@med.semmelweis-univ.hu

DOI: 10.14232/actacyb.23.2.2017.6

1 Introduction

1.1 Background

Today, telemedicine interoperability technology is applied in aeronautics, maritime and astronautics. Feasible telehealth information technology is a critical success factor for future planetary exploration programs as well: The ISS [International Space Station] and future planetary exploration-class missions (e.g., to Mars) will require the incorporation of up-to-date telemedicine concepts and technology, subject to the resource restraints and operational realities of space medicine [1]. Even though in this study we present results of land-based healthcare systems interoperability research, the conclusions reached here are quite general.

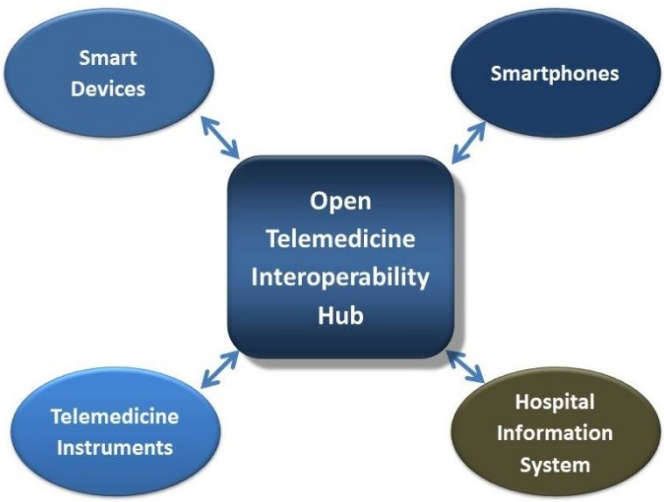


Figure 1: OTI-Hub Data-Link Diagram Schematics

Our research project commenced in 2015. We asked ourselves and tested whether there exists an information technology solution that allows interoperation among any biosensor enabled eHealth smart devices and any hospital information systems. Three technology sub-domains are specified in our program; namely eHealth smart device technology, telemedicine instruments technology and healthcare information system technology. If the hypothesis is proven, then in theory, there is no unresolvable technological obstacle preventing the interconnection of all eHealth smart devices, telemedicine instruments and hospital information systems. Furthermore, an extension of this would be in theory that all eHealth smart devices (eHSD), telemedicine instruments (TI) and healthcare information systems (HIS) could interoperate on a global scale. Here, current international healthcare interoperability standards and nomenclatures are taken into account as well. The crucial question for this research program is: can we build a (global) healthcare

ecosystem based on the Internet of Things framework? Our study is a synthesis of traditional healthcare information system architecture, the telemedicine instrument landscape, and mobile wearable bio-sensory technology. An integration solution, Open Telemedicine Interoperability Hub (OTI-Hub, Figure 1), relying on a state-of-the-art international healthcare interoperability standard, has been developed, put into operationimplemented and functionally evaluated by quality assurance procedures.

1.2 Research goal

The issue of interoperability among clinical systems, telemedicine instruments and eHealth smart device technology is currently unresolved. There is a need for the seamless technical and methodical interoperability among these three different healthcare domains because:

1. The emerging sensor-based smart devices collect and transmit large amount of data,

2. Sensor-based smart devices collect and transmit semi-continuous data series,

3. Clinical information systems transmit and store static data with reference to patient records,

4. The data representation, structure, method and rationale are different in hospital information systems than in eHealth smart devices.

Overall, the technical solution, bridging the usual clinical information systems and the eHealth smart devices, remains still a missing link.

This study attempts to establish an empirical basis for a universal between clinical systems and Internet of Things (IoT). Our purpose is to place the interoperability (represented by the OTI-Hub) in the center, and facilitate communication with all the above-mentioned domains (HIS, TI, eHSD) over the Cloud-based OTI-Hub. The key goal of the study is a design concept that meets this scientific and methodological need, if possible. Below we demonstrate the theoretical basis for this interoperability design concept. Based on it, the appropriate technical architectural landscape has been developed. Using this architectural concept the system components have been identified, defined and developed. The interoperability among the interconnected system components developed by us has been tested, documented and evaluated in the study. The technical evaluation was followed by clinical validation (Semmelweis University 2nd Department of Paediatrics Pulmonology Division) in order to ensure that the technical and clinical information obtained are both interpretable. This study intends to apply healthcare supply-chain interoperability factors beyond strictly defined information technology considerations, including applicability in everyday clinical workflows.

1.3 From a Healthcare Practitioners Point of View

Diabetes remains the number one global endemic disease. Insulin pumps developed for diabetes patients with sensors implanted under the skin require stable data-circulation. This solution supplemented with a Cloud-based interface and data-processing software transfers the actual value of the blood sugar level to the designated diabetes center via telecare and telemedicine software systems. The patients actual blood sugar information is displayed directly at the diabetes center, and the blood sugar timelines and periods are monitored and customized according to the patient's ailments.

The other significant group of endemic diseases is chronic asthma. Mobile telemedicine device developed for asthmatic patients is used by the diseased at his home and the results are automatically uploaded to the asthma clinical center through the Internet. The specialist analyzes the results during the patient's next visit or conducts a remote medical intervention, where necessary. The medical professional also evaluates the spirometry test results based on system standards.

Similar procedures are used in ECG tests and in the monitoring the the blood pressure monitoring of patients with cardiovascular disease.

Figure 2: Semmelweis University 2nd Paediatric Clinic Department of Pulmonology after the successful interconnection of an experimental tablet through dedicated WLan access to hospital information system

1.4 Research partnership

Finally, within the research project, the University of Debrecen, Faculty of Informatics, Department of Information Technology provided a good basis for ICT-related tasks. Secondly, the Semmelweis University 2nd Department of Paediatrics Pulmonology Division provided the medical background for the project (Fig. 2). Thirdly, an IT technology service provider at a healthcare competence center delivered the test system to the hospital in order to perform a series of experiments. The University of Debrecen and the Semmelweis University Clinic provide nearly ten percent of domestic healthcare coverage.

1.5 Experimental Laboratory Hospital Information System

Interoperability use cases were tested on a test-acceptance instance of the chosen hospital information system (MedSol). This hospital information system operates in sixty hospitals over Europe and serves forty thousand users. There are other international healthcare software providers on the market as well, including SAP Healthcare Solutions, IBM HospiLogix, Avicenna; Siemens Soarian Clinicals, GE Centricity and Oracle Healthcare Management Platform. However, the University Clinics of Debrecen and the Clinics of Semmelweis University have the necessary software licenses to run the MedSol system. The limited budget of the project did not allow us to test the OTI-Hub on other manufacturer's hospital information systems. Therefore, testing the HIS interoperability of different manufacturers through the OTI-Hub fell outside of the scope of our study.

1.6 Spirometry

The PDD-301/shm medical spirometer installed in out system represents the telemedicine instrument landscape in the research project. This medical telemedicine device is connected through the OTI-Hub, converted into Health Level Seven (HL7 [2, 3])-based interface files and transmitted to the hospital information test system.

A heart-rate monitoring smart wearable bracelet forms part of the eHealth smart device technology domain.

During the study the industrial telemedicine instrument interconnection was made at the Semmelweis University 2nd Department of Paediatrics Pulmonology Division, and the PDD-301/shm spirometer is connected to the clinical information system there (MedSol and eMedSol; figures 3 and 4). This system is accessed via tablets through local clinical WLAN by the medical staff. Our study well explain how the mobile spirometer and the also mobile clinical information system GUI cooperate with each other. This will be in effect a mobile telemedicine deployment simulation, like healthcare solutions for remote, sparsely populated regions like some cities in Norway, Sweden and Canada. Here, our study strictly applies the international HL7 healthcare interoperability standard.

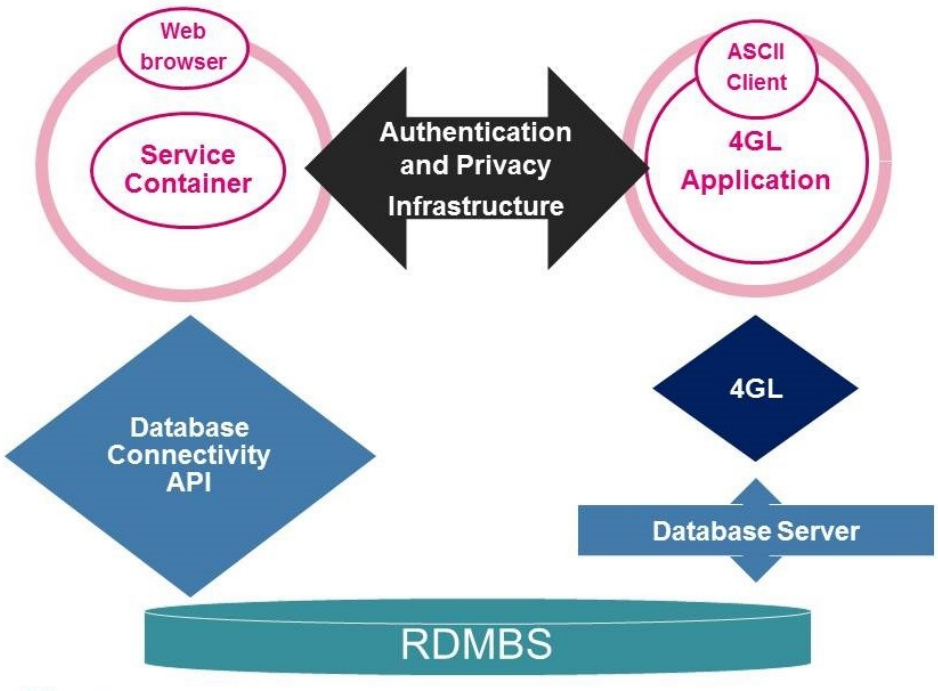


Figure 3: Typical Hospital Information System Schematic Architecture

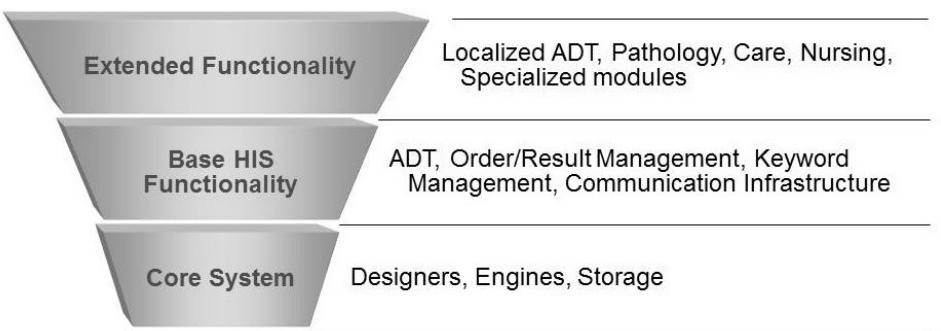


Figure 4: Modular-Hierarchical Hospital Information System Concept

1.7 Influential industry megatrends

Present IoT trends suggest that within a decade, billions of smart devices will be communicating nonstop with each other: Cisco estimates that by 2020, 50 billion devices and objects will be connected to the Internet [4]. As for the eHealth smart

device market, penetration should mean billions of new users will be included within the near future [4], so the healthcare information gathered so far is quite valuable. Industrial Big Data [5] analytical capabilities combined with incoming personal health data volumes open new horizons in personal and community healthcare forecasting including international epidemic control.

Global cloud infrastructure capabilities has reached its maturity over the past decade, and now they are ready to handle bulk sensory data flow. This allows us to develop Cloud-based interoperability solutions tuned for mass sensory-data handling, like the Open Telemedicine Interoperability Hub.

Cloud architectures are classified into private, public and hybrid clouds (community cloud is excluded in this study due to reliability and availability concerns). The private Cloud offers feasible technical solution for sensitive personal patient data; and the public Cloud delivers the necessary scalability.

Related Works are evaluated in Section 2. Section 3 determines the suitable syntactic healthcare systems interoperability standard and nomenclature. The clinical systems interoperability improvement proposal and standard recommendation is described in Section 4. The recommended service architecture topology, our design concept and deployed software technology is defined in Section 5. Section 6 stipulates the impact evaluation and technical research report including threats to validity. In Section 7, we discuss future work. Finally, in Section 8 we summarize our conclusions and views.

2 Related Works

A recent study has created a general software platform for clinical systems integration of telemedicine instruments [6]. However, these studies cited remain on a theoretical basis. Nevertheless, the available studies focus only on linking the telemedicine instruments with the mobile wearable body-sensory appliances.

All the cited literature treat the HIS and the IoT eHealth domain as two characteristically separated ecosystems. The relevant literature generally places the HIS at the center of the medical information technology landscape [7]. The studies in the field of interoperability focus either on the HIS-TI or on HIS-IoT interoperability. No comprehensive research has been found on handling the inter-domain (HIS-TI-IoT) interoperability.

2.1 Telemedicine instruments interoperability

Telemedicine instrument (TI)-hospital information system (HIS) interoperability has already been thoroughly studied and summarized in the relevant scientific literature [8, 9, 10]. International standards have been implemented regarding syntactic-level interoperability and healthcare data exchange (HL7, GDT, etc.). Process interoperability has also been thoroughly analyzed and scientists already have delivered proposals to solve the current gaps [11, 12]. However, regarding the TI-HIS interoperability, there is still plenty room for research in the area of non-

orthodox data exchange integration, like multilateral videoconferencing and diverse patient-information sharing (including, but not limited to transmission, capture and the interpretation of unstructured text and images) [13, 14].

2.2 Microcontroller-level interoperability

In the available literature microcontroller- and protocol-level generally applicable interoperability solutions are provided, like Sensor Hub and Sensor Hub Cloud Servers. Perera et al [15] describe their low-level interoperability solution with linkage to cloud-server processing. Lengyel et al present their general sensor hub framework [16]. Shibuta and Iwata demonstrates data-driven sensor hub architecture [17]. Patel and Sola exhibits a wireless sensor hub communicating through WiFi protocols [18]. These solutions all provide a partial solution to the healthcare interoperability problem. The results presented in this paper focus on Cloud-based syntactic and semantic interoperability across the telemedicine, IoT eHealth and classical healthcare domains. The current and cited Sensor Hub solutions do not cover the entire topic of this study, but provide a particular solution to the problem.

2.3 Consumer electronics and IoT eHealth interoperability

Kumar et al present an integration architecture suitable for IoT consumer electronics transmitting continuous data signals [19]. This study is a milestone, but it covers only a limited part of the problem of the actual research. Notwithstanding that for the Internet-of-Things (IoT) appliance-HIS interoperability has already been studied in the relevant international literature, these focus strictly on the given topic but unfortunately they do not offer a coherent architectural or a single standard solution [20, 21].

2.4 Regional healthcare interoperability

Barbarito et al describe the comprehensive regional healthcare information system implemented in the Lombardy region in Italy [22]. This system integrates the healthcare services for ten million citizens, 150000 health- and social care workers, 7800 general practitioners, 2600 pharmacies, 35 public hospitals, 15 local healthcare units and over 2500 private healthcare organizations. This project is a noteworthy landmark for general healthcare interoperability in practice. However, IoT eHealth device integration is not handled in this implementation project, and it is applied stand-alone classic servers instead of cloud technology.

2.5 Leading interoperability standards

The key healthcare electronic communication standards are the international Health Level Seven (HL7) and the German xDT standard family [23]. The xDT standard family consists of the GDT (Gertedatentransfer, instrument data transfer), BDT (Behandlungsdatentransfer, treatment data transfer) and LDT (Labor-

datentransfer, laboratory data transfer) data exchange format families. There are healthcare standardization organs like the USA-based National Council for Prescription Drug Programs (NCPDP, [24] and the also USA-based Organization for the Advancement of Structured Information Standards (OASIS, [25]). The Cloud Application Management for Platforms (CAMP, Standardizing cloud Platform-as-a-Service Application Programming Interface [26]) is a standardization initiative of the OASIS entitiy.

3 Applied Syntactic Healthcare Systems Interoperability Standard

The HL7 healthcare interoperability de facto international standard and the widely recognized SNOMED-CT [27] nomenclature have been chosen as the primary conditions for this study. The conceptual logic of the OTI-Hub system relies on the basis of the above mentioned standard and nomenclature. In theory, the vast majority of HIS and medical instruments are HL7-enabled [28]. However, in reality the HL7 standard itself has different, not fully compatible sub-versions (HL7 v2.x, HL7 v3). Notwithstanding that the latest HL7 v3 standard covers almost all internationally applicable medical information areas instead of the previous HL7 v2.x version [29], the HL7 v2.x is the de facto international interoperability standard for clinical information systems according to industry consultants. This is why we applied the HL7 v2.3 standard for the clinical spirometer-healthcare information system interconnection.

The HL7 v2 family consists of comma separated files (CSVs), while the HL7 v3 group has an XML structure. Transaction manager software products (like IBM Websphere and the MQ-Series) can handle both CSV and XML files successfully, truncate them when needed and forward the designated CSV or XML part to the recipient system or database table. Both CSV and XML files are applied in other heterogenous system landscapes, like global card transaction processing systems (e.g. VISA) and in billing systems (e.g. German Telekom Landline Billing).

This step (the application of the HL7 v2 instead of HL7 v3) provides a significant constraint. The recent HL7 v3 versions deliver a broad interoperability capability. However, our team was told by an industry consultant that the vast majority of the hospital information system deployment within Central- and South-Europe apply the older HL7 v2 version. This fact should have been taken into account in our study. The OTI-Hub itself is both HL7 v2 and v3 capable, but in the course of the testing the de facto HIS limitations influenced our project plan. The parametrized HL7 v3 output from the OTI-Hub is shown on Figure 5. It represents a fasting blood sugar (FSB) measurement after the patient had not eaten for at least eight hours. Usually the FSB is the first test to check for prediabetes and diabetes. The above-mentioned XML defines the accepted low and high values with a specified unit, but these are the extreme values. In this case, the specified unit is milligrams per deciliter. The measurement has an explicit value of 182 milligrams per deciliter. With this measurement, the specialists could diagnose

diabetes, because the measured value of over 105 milligrams per deciliter hit the defined maximum value.

```
<observationEvent>
   <id root="2.16.840.1.113883.19.1122.4" extension="1045813"
       assigningAuthorityName="GHH LAB Filler Orders"/>
   <code code="1554-5" codeSystemName="LN"
         codeSystem="2.16.840.1.113883.6.1"
         displayName="GLUCOSE^POST 12H CFST:MCNC:PT:SER/PLAS:QN"/>
   <statusCode code="completed"/>
   <effectiveTime value="200202150730"/>
   <priorityCode code="R"/>
   <confidentialityCode code="N"
      codeSystem="2.16.840.1.113883.5.25"/>
   <value xsi:type="PQ" value="182" unit="mg/dL"/>
   <interpretationCode code="H"/>
   <referenceRange>
      <interpretationRange>
          <value xsi:type="IVL_PQ">
          <low value="70" unit="mg/dL"/>
          <high value="105" unit="mg/dL"/>
        </value>
        <interpretationCode code="N"/>
      </interpretationRange>
   </referenceRange>
</observationEvent>
```

Figure 5: Parameterized HL7 v3 standard output from the OTI-Hub

The integration of the eHealth smart device technology within the classical medical system landscape is also necessary for the overall enhancement of the healthcare supply chain [30]. It establishes a link for traceable information exchange among conventional medical systems, industrial telemedicine instruments, eHealth smart devices and adaptive healthcare-services [31, 32]. The OTI-Hub system relies upon the international HL7 standard and provides bi-directional interoperability among a HIS, an eHSD and a TI [33, 34]. The OTI-Hub is embedded in a Cloud Architecture. This architecture brings significant benefits, but also takes into account patient data privacy issues.

4 Clinical System Interoperability Improvement Proposal and Standard Recommendation

Our goal is to create a flexible telemedicine interoperability hub that will extend the options of conventional hospital systems. To achieve this goal, industry-wide accepted technologies have been applied [35, 36]. Also, the software development environment for the hub has been selected and made concrete (the selected technologies are available on Microsoft stack, and the OTI-Hub system has been im-

plemented with Microsoft stack and other open source technologies [37]).

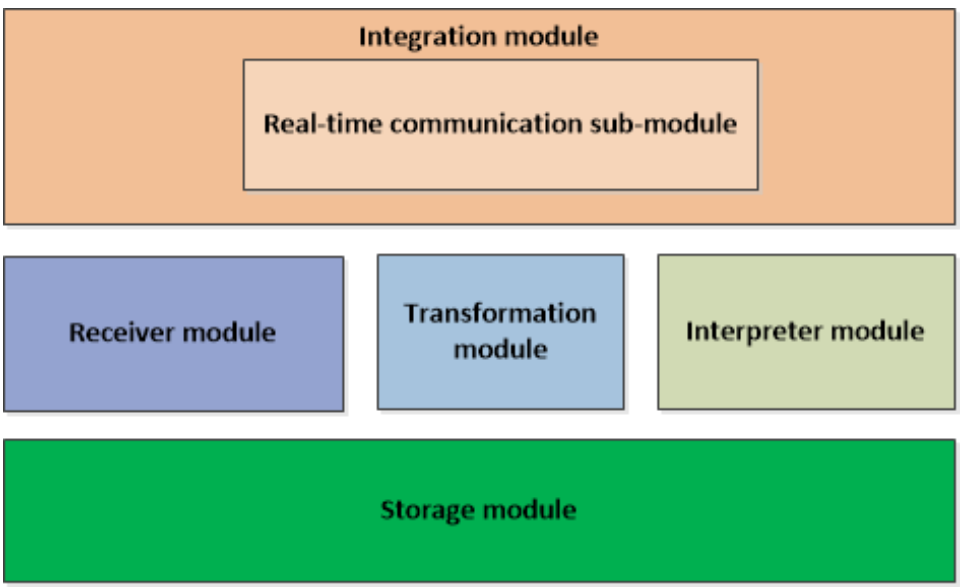


Figure 6: OTI-Hub Service Architecture

The following technologies were applied within the OTI-Hub (Figure 6) and in the modules:

The receiver module handles the received measurement data through http Web request. This module uses a Web API library and associated markup language (represented by Microsoft Web API, JSON and XML in the study). This module applies an open authentication protocol (OAuth in this study) to authorize and authenticate the users and devices. The REST principles also need to be adapted, where possible [38, 39].

The transformation module operates on the data collected by the receiver module and the main task of this module is to transform the date to the chosen interoperability standard format (HL7 and C#-based Windows service in the study). This module is of course a critical part of the system. As eHSD manufacturers specify different output formats, the transformation to a unified format is a cornerstone for interoperability. The key to the OTI-Hubs general expansibility is an incoming and outgoing data format conversion. At the moment the transformation module is the OTI-Hubs internal component. An independent transformation module should be planned, built and deployed later, so as to make the Hub generally applicable.

The data storage module is responsible for building a data warehouse from the data that was collected and transferred. The OTI-hub uses the storage part of Apache Hadoop, known as the Hadoop Distributed File System (HDFS), to store the measurement data. It is a file based storage solution, where the Hadoop splits the files into very large blocks and distributes them across the nodes in the cluster.

The interpreter module works with the data warehouse module and its main task is to interpret the data that was collected and aggregated. The OTI-hub uses the Apache Hadoop software library to evaluate the measurement data. Apache Hadoop utilizes the MapReduce programming model. It transfers package codes into cluster nodes to process the data in parallel. This approach takes advantage of data and file locality, every node manipulating the data and files they have access to.

Here, the key module is the *Hubs integration module.* This module sends export data features from the telemedicine hub to external systems like hospital information systems. This module uses REST API endpoints to transport data to external systems; and it applies the technologies summarized in the receiver module (Microsoft Web API, OAuth, JSON and XML in the study [40]).

The real-time Web communication sub-module library operates in order to maintain an open socket between the Hub and the devices (SignalR in our study). This module is critical when a device is recording data frequently and the received data have to be readily availabe on the Hub. In this case, when lower priority measurement data are not needed immediately on the Hub, this module can be excluded from the process chain. The socket allows one to use a channel between the Hub and the device without reconnection and re-authentication. The only significant latency is network latency in this case. The OTI-Hub is suitable for a full duplex channel, so the data can be transmitted and received in both directions.

The Hub is embedded in a Hybrid Cloud Architecture. The Hub component, responsible for the patient-related master data, is embedded in a private cloud architecture (German Telekom Private Cloud in our study). The Hub-components, responsible for the interoperability logic and routing, are implanted in a public Cloud (the Google Cloud Platform in our study).

Here, the solution presented here is an open architectural solution providing the basis for healthcare interoperability. It is open for the future expansion and inclusion of new specific technologies. Based on the Hub's internal structure, it can be adjusted to varying incoming formats. By reparametrizing the Hub, it can process other custom protocols as well (e.g. bio-sensory dataflows forwarded by cell-phones).

5 The Recommended Service Architecture Topology and Software Technology

5.1 The Service Architecture Topology

Here, our aim is to set up a service architecture through the OTI-Hub that will enable bi-directional syntactic interoperability among HIS, TI and eHSD. Therefore, we established a cloud-based system landscape (Figure 7) based on the OTI-Hub and the HL7 interoperability standard.

The TI is interconnected with a personal computer (PC) via a USB. The factory client program runs on this PC. This is interconnected by a Wide Area Network

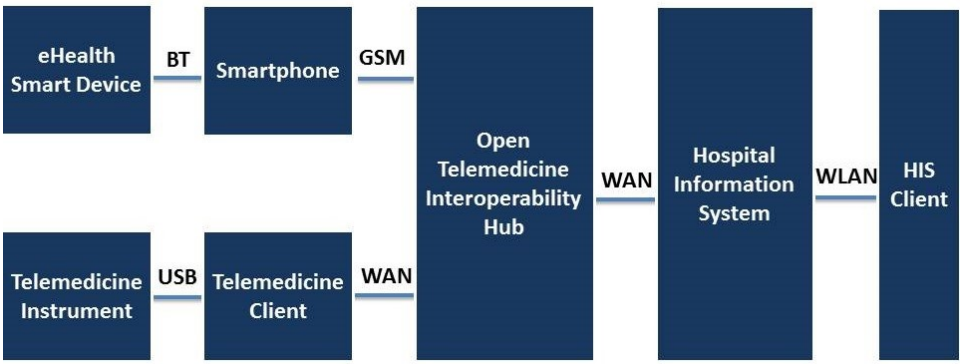


Figure 7: Telemedicine and eHealth Systems Interoperability Landscape with OTI-Hub

Internet connection to the cloud-based OTI-Hub. The eHSD communicates via a Bluetooth connection to the smartphone, which in turn runs the OTI-Hub App. This App, developed by our research team may be downloaded and installed by the Windows App Store to this device. The smartphone connects via a GSM or W-Lan internet connection to the OTI-Hub. The OTI-Hub is connected via a WAN Internet-connection to the HIS. The prototype tablets are connected by a dedicated internal W-Lan connection to the HIS, and the HIS online Graphical User Interface (GUI) runs in the tablet's browser.

The novelty of this architecture is that it integrates all the conventional healthcare information system architecture, the standard telemedicine environment and the eHSD technology.

The cloud-based Healthcare Information System (HIS) we applied, called eMedsol currently operates in sixty hospitals and serves forty thousand users in Hungary, Romania, the Czech Republic and Bosnia-Herzegovina. The eMedSol system is equipped with a UNIX-based (Linux, AIX HP-UX, SCO) WebSphere Application Server (V5, V6) that is connected to Oracle (10gR2) and Progress (V10 OpenEdge) relational database management systems (RDBMS). Each installation (for a clinical institution with fewer than three hundred beds) is equipped with a primary and a secondary virtual server, two quad-CPUs and 16 GB Memory. The OTI-Hub is embedded in a Google public Cloud (Google Cloud Platform), and sensitive patient personal data are stored in the Mnchen-based Open Telekom Cloud datacenter to ensure that patient data remain within the EU.

The service architecture shown in Figure 8 provides the methodological basis for the kind of IT solution (e.g. OTI-Hub) that provides interoperability among the conventional healthcare IT components, telemedicine appliances systems and eHSD device technology.

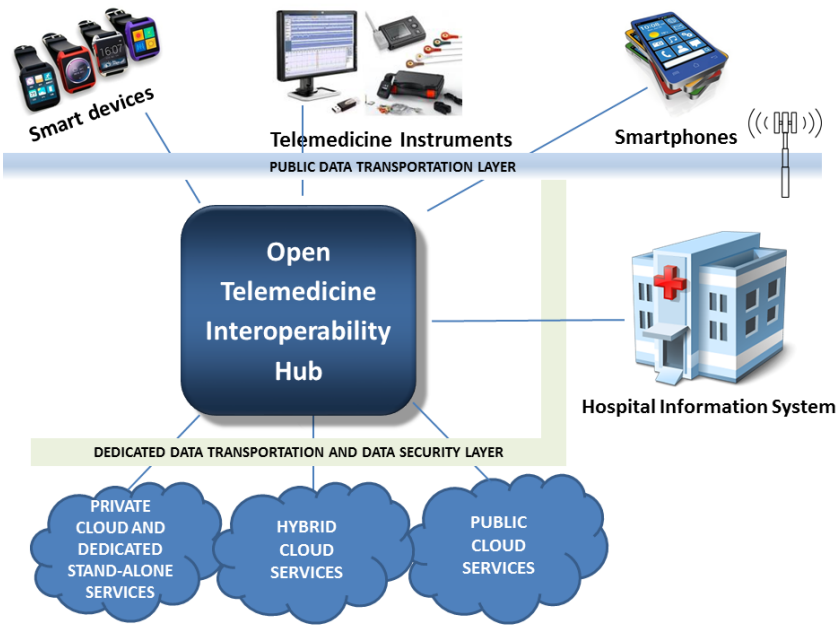


Figure 8: Our Open Telemedicine Interoperability Hub (OTI-Hub)

5.2 Software Technology

The concept of clinical systems interoperability goes far beyond simple data exchange: it constitutes interoperability at the technical, semantical and process levels. In an empirical model of the study the OSI model (ISO/IEC 7498-1:1994 [41]) is mapped against these interoperability levels, hence these three interoperability modalities can also be interpreted at the corresponding IT abstraction layer. In our study we focus on both technical and semantic interoperability. Among the technical interoperability modalities, instead of the TCP/IP, we chose the file-based interface connection, since this option offered significantly more flexibility during our study [42, 43, 44]. The clinical spirometer PDD-301/shm was linked over a USB connection to the corresponding spirometry desktop program provided by the manufacturer.

The following instruments were selected and allocated to the study program: the spirometer PDD-301/shm as a clinical telemedicine instrument, Microsoft Band I and Microsoft Band II smart wristbands as eHealth sensory devices, a Nokia Lumia 930 smartphone, the Windows 10 Mobile operating system, the Dell Latitude E6520 (Windows 10, 32-bit operating system, i5-2520M chipset, 4 GB RAM and 256 GB HDD) primary laptop, a Dell Latitude E6220 (Windows 7 64 bit operating system, i5-2520M chipset, 4 GB RAM, 128 GB SDD) secondary laptop, three Lenovo MIIX 300-10IBY tablets and an ACER SWITCH SW3-013-12CD tablet.

Each tablet was equipped with a 10,1 display (WXGA and HD IPS), 2 GB memory, 64 GB internal storage and Windows 10 operating system. All the laptops and tablets were compatible with the 802.11g WLAN and Bluetooth 4.0 standards; and the spirometer is USB-enabled. The chosen smart wristbands were manufactured with built in- Bluetooth 4.0 communication chipsets. Each instrument of the lab equipment package was also individually tested prior to the experiments.

A specific private cloud was set up for our study. This ran on a stand-alone x86-64 architecture equipped with an Intel i5 processor, 256GB SSD and 4 GB RAM. The operating system for the private Cloud was Red Hat Enterprise Linux 7.0 3.10.0-229, the virtualization was provided by VMware Workstation v6.5.0 and the relational database management system was supplied by MySQL v5.6. The cloud-based version of the hospital information test system ran in a commercial private cloud (Telekom Cloud). The Open Telemedicine Interoperability Hub data transmission module was embedded in a commercial public Cloud called Microsoft Azure.

The HIS ran on J2EE WebSphere Application server V6, which relied on Oracle RDBMS 10gR2 and Progress V10 OpenEdge RDBMS. Also, the HIS was hosted on the Unix operating system. Floating licenses were made available for accessing the online, cloud-based edition of the HIS via the experimental tablets. The Open Telemedicine Interoperability Hub development environment consisted of the Universal Windows Application Development Tools (1.4.1), Windows 10 Software Development Kit 10.0.25431.01 Update 3 and Microsoft .NET Framework Version 4.6.01038. The OTI-Hub internal database was developed using the SQL Server Data Tools 14.0.60519.0. The OTI-Hub App was developed using Visual Studio Tools for Universal Windows Apps 14.0.25527.01. The OTI-Hub middleware was embedded in Microsoft Azure Mobile Services Tools 1.4. Red Hat Enterprise Linux 7.0 3.10.0-229 provided the operating system for the private cloud set up specifically for the study.

The spirometry desktop program was installed on a standalone Dell Latitude E6520 laptop equipped with the Windows 10 operating system. The spirometer was calibrated by the manufacturer for the study. A forced vital capacity spirometry test was performed with a healthy individual. After the test results had been stored in the spirometry desktop software package, the HL7 v2.3.1 interface file was exported. This interface file was processed by the cloud-based OTI-Hub. The OTI-Hub appended the spirometry information with the earlier transformed cardio body-sensor information captured by the L18 Smart Bluetooth Wristband. The generated HL7 interface file was imported after parameterization into the factory acceptance test instance of the MedSol hospital information system. Both the imported spirometry and cardio test results were retrieved and displayed by the patient report query of the hospital information system.

The OTI-Hub is embedded in a hybrid-cloud. The OTI-Hub patient-data-related components ran within the Private Cloud Infrastructure-as-a-Service environment. The other OTI-Hub components ran within the public cloud Platform-as-a-Service environment.

6 Impact Evaluation and Technical Research Report

The IT results were validated by the Department of Information Technology, University of Debrecen and by our Industry Partner. The clinical results were validated by the Semmelweis University 2nd Department of Paediatrics.

The spirometry (see Figure 9) and cardio sensory HL7 test result data transferred via the OTI-Hub was successfully imported, interpreted and presented in the target HIS. As expected, the spirometry and cardio test results were correctly reflected through the HISs patient result query. The cardio information in the HIS query displayed the values transformed by the OTI-Hub. The results told us that HL7-based health data interchange among different Information Technology architectures was fully feasible. The OTI-Hub successfully provided seamless interoperability among Android-, UNIX- and Windows-platforms. Both cloud and standalone architecture components were also effectively interconnected during our study.

The HL7-based information exchange among the healthcare information acceptance system, the dedicated standalone spirometry system (see Figure 10) and the OTI-Hub was stable and trackable (see Figure 11). The combination and compatibility of the different HL7 versions were mainly handled within the OTI-Hub logic. The conversion of the body-sensory smart device output data stream into meaningful HL7 interface information was a significant challenge. While the spirometry output interface file was sent and processed by the OTI-Hub correctly, healthcare smart-device manufacturers unfortunately do not provide well-specified output format in most cases. These smart device output formats are typically readable for humans, but not precise enough for automatic processing [43].

In addition, the healthcare smart device output data streams are not ready to be processed by HIS unless they can be transformed into meaningful static values. However, our OTI-Hub successfully imported and interpolated the randomly selected healthcare smart devices (smart bracelet) data stream into interpretable HL7 values in this particular case. Naturally, this issue will require further investigation. As healthcare smart devices coming from different manufacturers use a very diverse output data-format and data emission frequency, there is room for overall standardization here to achieve a single format. It would be most desirable to find a solution for this issue. However, this topic goes far beyond the technical aspects: commercial interests, patent rights and other non-technical considerations influencing this area. From a technology perspective, sensor-Hub technologies, international biosensor-flow standards and best practices could deliver the desirable solution for this issue. However, finding a good solution for an industry-wide standardized primary bio-sensory data-format falls outside the scope of this study. To sum up, the actual results of the study meet the requirements, and provided significant useful lessons for us, as we suggested in the consclusions section (see Section 8).

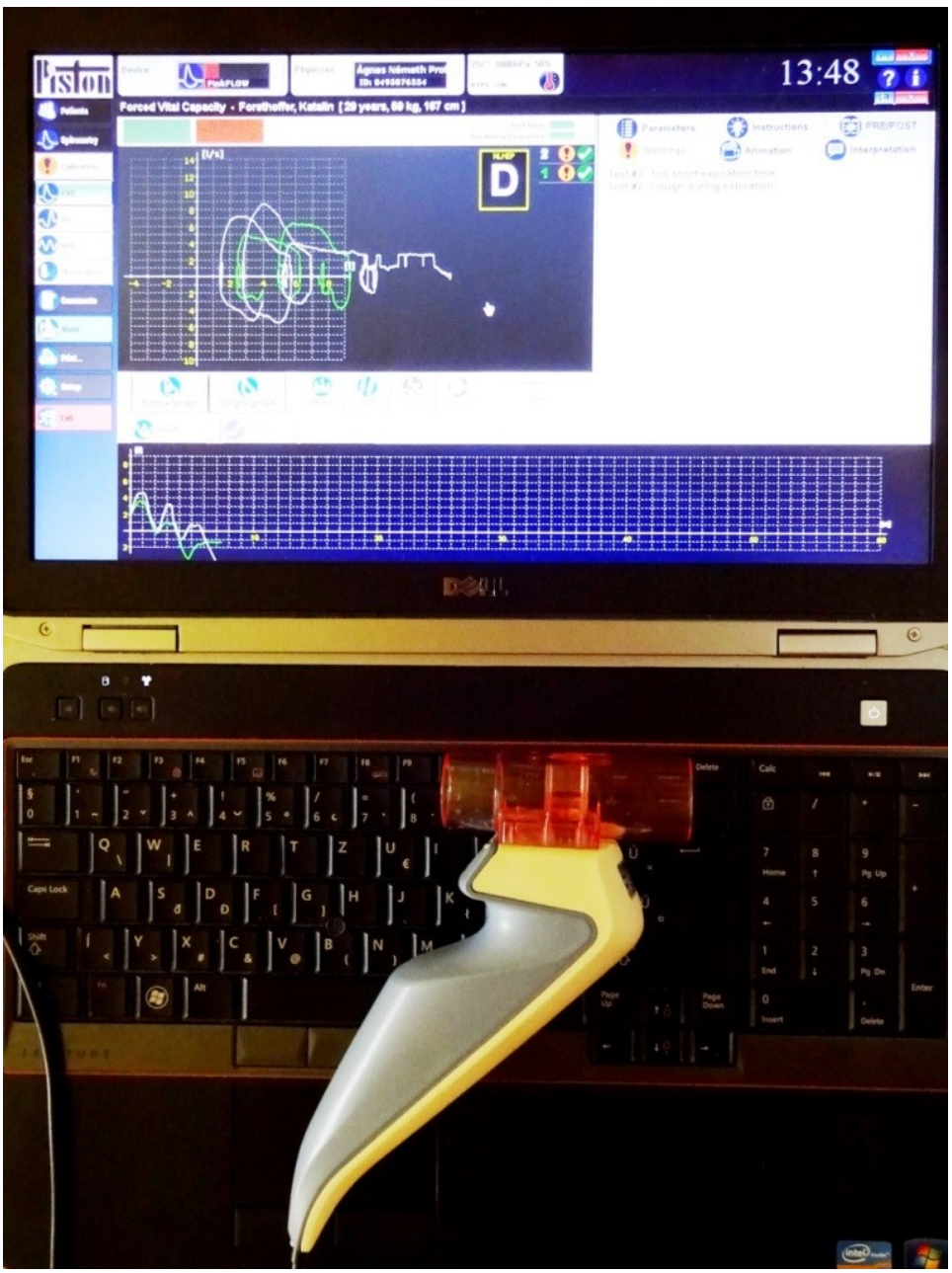


Figure 9: Data-interchange over our OTI-Hub with a clinical spirometer PDD-301/shm

Figure 10: A Spirometry Clinical Test Result data-export to the OTI-Hub in HL7 v2.3.1 format

```
MSH|^~\&|PXP||||20160802095510||ORU^R01|20160802095510|P|2.3.1|||NE|AL|HUN|
PID||65488965|18||Patient^Anonymised^^^^Mr.||19870513|F|||||||||||||||||
OBR|1||20160802094856^PXP|94011|||20160802094856||||||||||||||2016080209485
OBX|1|ST|0^FVC^99MKW||3,29|l|3,75||||F
OBX|2|ST|2^FEV*0.5^99MKW||1,04|l|||||F
OBX|3|ST|3^FEV*1.0^99MKW||1,57|l|3,27||||F
OBX|4|ST|5^FEV*0.5/FVC^99MKW||31,70|%|||||F
```

Figure 11: An extract from the Clinical Spirometer HL7 v2.3.1 Output File got via the OTI-Hub

6.1 Threats to validity

Our study did not analyze the data-flow in the backward direction, e.g. when data originating from a conventional HIS are sent to a healthcare smart app running on smartphone using the OTI-Hub. This backward healthcare information flow should be examined in a later study. Here, ours applied file-based interface connectivity. The data package based interconnection may react differently under real-life conditions, and it may have its advantages (speed) and disadvantages (blocked communication in the case of a broken link in the dataflow-chain). The commercially available eHSDs do not meet healthcare standards; therefore the intercepted data may not be clinically as precise as it is required (see Figure 12 and Figure 13).

Even global cloud providers have different regional datacenters and the results in geographically different levels of service. This will limit the performance of the OTI-Hub at the international level.

Figure 12: The smart device used in our study

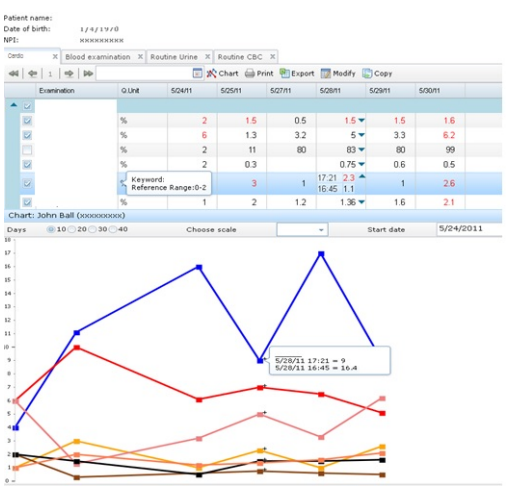


Figure 13: eHealth Smart Wearable Device Communicating via the OTI-Hub

7 Future Work

The key purpose of the study was to create and implement a hybrid Cloud-based prototype that provides clinical research patient data using a smart-device body-

sensory information flow. Here, we asked ourselves whether it was possible that body-sensory smart device technology and telemedicine instrument architecture could be integrated into the classical healthcare information landscape. Now, it is clear that the answer is categorical affirmative.

Based on the results and experience gained by our research, we see that it is indeed possible to achieve the flawless interconnection of the body-sensory IoT smart device technology, the telemedicine architecture and the conventional healthcare IT landscape. Our study indicates that meaningful healthcare information exchange is possible among HL7 capable HIS and eHSDs. Another important point is that the interconnection of different platforms (UNIX, Windows) and manufacturer-specific smart devices requires special nesting software classes to handle the slightly different coding conventions and data formats. The proposed API provides a bridge between these systems and devices by specific conversions and it provides a consistent way to access all the data originating from the different parties.

The full-duplex bi-directional interoperability will constitute the next area of our research. Here, we demonstrated unidirectional simplex interoperability over the OTI-Hub. In the next phase, bi-directional functionality will be set up and evaluated. The Hub's transformation module will be relocated as an independent component outside of the Hub. This reshaped transformation module will also run in an Android app-based java class. This app itself will also transform the incoming bio-sensory dataflow into a unified data format. As a result, the OTI-Hub will be extended to interpret various incoming sensory-data acquired from different sources, as long as it is transformed into a unified format by the preloading module.

An evaluation on the interoperability of two different software manufacturer's HIS' via the OTI-Hub is planned in the next phase of our research. Telecommunication infrastructure backbones preparedness (e.g. GSM Network Bandwidth) should also be assessed in the future for securing a substantial increase in the IoT data-exchange volume.

8 Conclusions

As the reader, can see our OTI-Hub described above serves as a methodological basis and software solution for the interconnection of the world of IoT and conventional healthcare technology based on standards. We successfully demonstrated and tested the interoperability built on our hybrid cloud-based architectural solution with the OTI-Hub. The OTI-Hub provides an open architectural solution for solving the interoperability issue among HIS, eHSD and TI domains. Here, the data transformation into unified standard is critical for the extensibility of the presented solution. I should be added that healthcare smart device technology will generate an unprecedented amount of human body-sensory data and this will be made available in the foreseeable future. This bulk information could serve inter alia as important input for epidemic control and set new targets for pharmaceutical development. Big Data analytics methodologies foster pattern and trend analyses based on the captured body-sensory healthcare information base and offer a new

way for crowd-sourced information handling. Our proposed hybrid cloud architecture assures the essential scalability for the OTI-Hub with its required robust transaction processing capacity. The architectural topology and systems integration solution illustrated here provides a technological solution for the integration of bi-directional international body-sensory, telemedicine and conventional healthcare data exchange. The results presented here offer some optimism, but current national healthcare data-related legal prerequisites need to be internationally harmonized for the required breakthrough. Our own OTI-Hub solution provides international eHealth data-exchange capability. Furthermore, the cloud-based eHealth interoperability solution presented here offers a framework for other application areas like the cloud-based implementation of the da Vinci Surgical System.

References

[1] Simmons S. C., Hamilton D. R., McDonald P. V. Principles of Clinical Medicine for Space Flight. Springer New York, New York, 2008, p. 163.

[2] ISO/HL7 10781:2015, HL7 Electronic Health Records-System Functional Model, Release 2, HER FM. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57757` (last visited on 16 September in 2016)

[3] Health Level Seven Standard Version 2.3.1 An Application Protocol for Electronic Data Exchange in Healthcare Environments, Health Level Seven International `http://www.hl7.org/` (last visited on 3 March in 2017)

[4] World Development Report 2016: Digital Trends. World Bank Group, IBRD (The World Bank), Washington DC, 2016, p. 328 (354). `http://documents.worldbank.org/curated/en/896971468194972881/pdf/102725-PUB-Replacement-PUBLIC.pdf` (last visited on 1 July in 2016)

[5] Definition of Big Data, Merriam-Websters Collegiate Dictionary, Web. `http://www.merriam-webster.com/dictionary/big%20data` (last visited 4 July in 2016)

[6] Zarour, K. Proposed technical architectural framework supporting heterogeneous applications in a hospital. International Journal of Electronic Healthcare, 2016, Vol. 9, Issue 1, pp. 19-41.

[7] Warren, S., Craft, R. L., Parks, R.C., Gallagher, L. K., Garcia, R. J., Funkhouser, D. R. A proposed Information Architecture for Telehealth System Interoperability. Conference: Toward an Electronic Patient Record 99, Orlando, USA, 2-6 May, 1999, pp. 1-10.

[8] Gaynor, M. G. Evaluation of Patient to Provider Oriented Telemedicine in Hospitals and Physician Practices. Muskie School Capstones, Paper 103, 2015, pp. 1-32.

[9] Fuhrman, S. A., Lilly, C. M. ICU Telemedicine Solutions. Clinics in Chest Medicine, Elsevier, Vol. 36, Issue 3, 2015, pp. 401-407.

[10] Emery, S. Telemedicine in Hospitals Issues in Implementation. Garland Publishing, New York, 1998.

[11] Kuziemsky, C. E., Peyton, L. A framework for understanding process interoperability and health information technology. Health Policy and Technology, Vol. 5, Issue 2, 2016, pp. 196-203.

[12] Bouamrane M. M., Tao C., Sarkar I. N. Managing Interoperability and Complexity in Health Systems. Methods Inf Med, Issue 54, 2015, pp. 1-4.

[13] Wei-Li L., Kai Z., Craig L., Michael A. Cloud and Traditional Videoconferencing Technology for Telemedicine and Distance Learning. Telemedicine and e-Health, Vol. 21, Issue 5, 2016, pp. 422-426.

[14] Taylor, A., Morris G., Tieman J., Currow D., Kidd M., Carati C. Building an Architectural Model for a Telehealth Service. E-Health Telecommunication Systems and Networks, Vol. 4, No. 3, 2015, pp. 35-44.

[15] Perera C., Jayaraman P., Zaslavsky A., Christen P., Georgakopoulos D. Dynamic Configuration of Sensors Using Mobile Sensor Hub in Internet of Things Paradigm. CoRR, Vol. abs/1302.1131, 2013.

[16] Lengyel, L., Ekler P., Ujj T, Balogh T., Charaf, H. SensorHUB: An IoT Driver Framework for Supporting Sensor Networks and Data Analysis. International Journal of Distributed Sensor Networks, Hindawi Publishing Corporation, Vol. 2015, 454379, 2015.

[17] Shibuta, H., Iwata, M. Self-Timed I/O Architecture of Data-Driven Sensor Hub. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Athens, 2016, pp. 323-328.

[18] Patel H. R., Sola R. Implementations of Wireless Sensor Hub to Support Protocols Interoperability. International Journal of Research in Computer and Communication Technology, Vol. 4, Issue 4, March 2015, pp. 192-197.

[19] Gay V., Leijdekkers, P. Bringing Health and Fitness Data Together for Connected Health Care: Mobile Apps as Enablers of Interoperability. Journal of Medical Internet Research, 17(11) e260.

[20] Nijeweme-dHollosy W. O., van Velsen L., Huygens M., Hermens H. Requirements for and barriers towards interoperable eHealth technology in primary care. IEEE Internet Computing, Vol. 19, Issue 4, July-August, 2015, pp. 10-19.

[21] Kumar R. B., Goren N. D., Stark D. E., Wall D. P., Longhurst, C. A. Automated integration of continuous glucose monitor data in the electronic health records using consumer technology. Journal of the American Medical Informatics Association, Vol. 23, 2016, pp. 532-537.

[22] Barbarito F., Pinciroli F., Mason J., Marceglia S., Mazzola K., Bonacina S. Implementing standards for the interoperability among healthcare providers in the public regionalized Healthcare Information System of the Lombardy Region. Journal of Biomedical Informations, Vol. 45, 2012, pp. 736-745.

[23] xDT, European Innovation Partnetship, European Commission, Standards, IcT and communication. `https://ec.europa.eu/eip/ageing/standards/ict-and-communication/other-ict/xdt_en` (last visited on 20 January 2017)

[24] National Council for Prescription Drug Programs. `http://www.ncpdp.org/` (last visited on 20 January in 2017)

[25] Organization for the Advancement of Structured Information Standards, OASIS. `https://www.oasis-open.org/` (last visited on 30 January 2017)

[26] Cloud Application Management for Platforms (CAMP) TC. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp` (last visited on 30 January in 2017)

[27] International Health Terminology Standards Development Organisation, Systematized Nomenclature of Medicine Clinical Terms (SNOMED-CT) `http://www.ihtsdo.org/snomed-ct` (last visited on 20 August in 2016)

[28] Fong B., Fong A. C. M., Li C. K. Telemedicine technologies: Information technologies in medicine and telehealth, Chichester, Wiley, 2011.

[29] Shaikh, A. The impact of SOA on a system design for a telemedicine healthcare system. Network Modeling Analysis in Health Informatics and Bioinformatics, 4:15, Springer Vienna, Vienna, 2015, pp. 1-16.

[30] Eren H., Webster J. G. The e-medicine, e-health, m-health, in Telemedicine and Telehealth Handbook, Oakville, CRC Press, 2015.

[31] Adenuga, O. A., Kekwaletswe, R. M., Coleman, A. eHealth integration and interoperability issues: towards a solution through enterprise architecture, Health Information Science and Systems, Vol. 3, Number 1, 2015, pp. 1-8.

[32] Luz, M. P., de Matos Nogueira, J. R., Cavalini, L. T., Cook T. W. Providing Full Semantic Interoperability for the Fast Healthcare Interoperability Resources Schemas with Resource Description Framework. Healthcare Informatics (ICHI), 2015 International Conference on, Dallas, 21-21 October 2015, pp. 463-466.

[33] Garai A., Pentek I. Adaptive services with cloud architecture for telemedicine, 6th IEEE Conference on Cognitive Infocommunications, Gyor, Hungary, 19-21 October, 2015, pp. 369-374.

[34] Adamko A., Garai A., Pentek I. Common open telemedicine hub and infrastructure with interface recommendation, 11th IEEE International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 12-14 May 2016, pp. 385-390.

[35] Garai A. Methodology for assessment validation of platform migration of robust critical IT-systems, 8th International Conference on Applied Informatics, Eger, Hungary, 27-30 January 2010, pp. 445-448.

[36] Adamko A., Garai A., Pentek I. Common open telemedicine hub and interface standard recommendation, The 10th Jubilee Conference of PhD Students in Computer Science, Szeged, Hungary, 27-29 June 2016, pp. 24-25.

[37] Adamko A., Kollar L. A system model and applications for intelligent campuses: Intelligent engineering systems, 18th International Conference on Intelligent Engineering Systems, Tihany, Hungary, 3-5 July 2014, pp. 193-198.

[38] Baranyai P., Csapo A. Definition and synergies of cognitive infocommunications, Acta Polytechnica Hungarica, Vol. 9, No. 1, 2012, pp. 67-83.

[39] Adamko A., Arato M., Fazekas G., Juhasz, I. Performance evaluation of large-scale data processing systems, Proceedings of the 7th International Conference on Applied Informatics, Eger, Hungary, 28-31 January 2007, Vol. 1, pp. 295-301.

[40] Adamko A., Kollar L. MDA-based development of data-driven Web applications, Proceedings of the Fourth International Conference on Web Information Systems and Technologies, Funchal, Madeira, Portugal, 4-7 May 2008, Vol. 1, pp. 252–255.

[41] ISO/IEC 7498-1:1994 Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model (OSI-Model), International Organization for Standardization (ISO), Web. 6 June 2016. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=20269` (last visited on 15 July 2016)

[42] Szabo R. et al, Framework for smart city applications based on participatory sensing, IEEE 4th International Conference on Cognitive Infocommunications, Budapest, Hungary, 2-5 December 2013, pp. 295-300.

[43] Pandi, K., Charaf, H. Mobile Resource Management Load Balancing Strategy. Acta Cybernetica, Vol. 22, Number 1, Szeged, 2015, pp. 171-181.

[44] Giachetta R., Fekete I. A Case Study of Advancing Remote Sensing Image Analysis. Acta Cybernetica, Vol. 22, Number 1, Szeged, 2015, pp. 57-79.

Acta Cybernetica **23** (2017) 537–559.

Automatic Calculation of Process Metrics and their Bug Prediction Capabilities

Péter Gyimesi[a]

Abstract

Identifying fault-prone code parts is useful for the developers to help reduce the time required for locating bugs. It is usually done by characterizing the already known bugs with certain kinds of metrics and building a predictive model from the data. For the characterization of bugs, software product and process metrics are the most popular ones. The calculation of product metrics is supported by many free and commercial software products. However, tools that are capable of computing process metrics are quite rare. In this study, we present a method of computing software process metrics in a graph database. We describe the schema of the database created and we present a way to readily get the process metrics from it. With this technique, process metrics can be calculated at the file, class and method levels. We used GitHub as the source of the change history and we selected 5 open-source Java projects for processing. To retrieve positional information about the classes and methods, we used SourceMeter, a static source code analyzer tool. We used Neo4j as the graph database engine, and its query language - *cypher* - to get the process metrics. We published the tools we created as open-source projects on GitHub.

To demonstrate the utility of our tools, we selected 25 release versions of the 5 Java projects and calculated the process metrics for all of the source code elements (files, classes and methods) in these versions. Using our previous published bug database, we built bug databases for the selected projects that contain the computed process metrics and the corresponding bug numbers for files and classes. (We published these databases as an online appendix.) Then we applied 13 machine learning algorithms on the database we created to find out if it is feasible for bug prediction purposes. We achieved F-measure values on average of around 0.7 at the class level, and slightly better values of between 0.7 and 0.75 at the file level. The best performing algorithm was the *RandomForest* method for both cases.

Keywords: process metrics, graph database, bug prediction

[a]Department of Software Engineering, University of Szeged, Hungary,
E-mail: pgyimesi@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.7

1 Introduction

Nowadays, companies tend to spend a large amount of resources on debugging and fixing software faults. Predicting these bugs can greatly help to reduce the costs. For this reason, bug prediction has become a popular research area. Recognizing bug-prone source code parts requires that one characterize them in some way.

There are many good studies on bug characterization [6, 22, 3, 4]. It can be carried out with classic product metrics, with software process metrics or with some metrics of a different nature like textual similarity. Product metrics are extracted from the structure of the source code. Some examples are lines of code, cyclomatic complexity and number of methods. There are many tools – some of them are free – which can produce these metrics for projects of different programming languages. These metrics are frequently used for bug characterization [19], because it is easy to compute them. Product metrics depend on a single state of the software and no project history is required; thus no temporal characteristics are used. These metrics are usually computed for files or classes, but an increasing number of tools support methods too.

Software process metrics are computed from developer activities. Most of them include some kind of temporal information. The most common ones are based on the number of previous modifications, number of different contributors, number of modified lines and the time of the modifications. These metrics can of course be used for a variety of purposes. Since the computation is based on the developers' activities, these values are perfect for examining the developers' behavior. Furthermore, locating key source code parts that are modified often or recently is another possible utilization of these metrics.

Previous studies [20, 16, 10, 11] have shown that while software process metrics are generally better bug predictors than product metrics, tools that can compute these metrics are still quite rare. The studies focus mainly on the definition of process metrics and the results. The method of computing these metrics is not always described, so reproducing these results may be a challenging task. It may be due to the difficulties of storing and processing the historical information. An important criterion here is to have available project history. Versioning systems (like Git or Subversion) are commonly used in software development, thus the history of a project is quite often accessible through an API. Source code hosting services like GitHub or Bitbucket are becoming evermore popular and contain open-source projects of various programming languages. Another criterion is that the developers have to use this system correctly, otherwise this information is not useful and it may be misleading.

The first problem we run into is the size of this data set. A project may contain hundreds of thousands lines of source code and also thousands or tens of thousands of commits. To compute the process metrics for one software version, the whole history has to be processed, hence an efficient method is required for this task.

Another aspect of this problem is the granularity of process metrics. Calculations can be made at different levels: file, class or method. At the file level, it is fairly simple, because versioning systems work with files, so no additional analysis

is needed. However, at the class level and the method level, a thorough source code analysis is required to extract the source code elements and their position. For more accurate results, other information (empty lines, comments, etc.) may be gathered. This task can be carried out with a static source code analysis tool[1].

The next issue is how to store the gathered data in an easily accessible form. In the past few years, the popularity of graph databases has increased due to the improving technologies behind them. A graph database can handle a large amount of data and it is suitable for storing weakly structured data. The historical data of a software package can be represented as a graph, so studies [2] have started to examine the application of these graph databases to assess software quality, especially in the calculation of software process metrics, hence graph databases seem to be a good choice for this task.

We chose GitHub as a data source because it contains more than 38 million repositories[2] and has a readily usable API[3] to access these projects. Furthermore, these repositories are accessible via Git[4]. We chose SourceMeter[5] as a source code analysis tool, because it is capable of processing five programming languages (Java, C++, C#, Python, RPG) and it can extract detailed information about the source code elements, including methods. The results of this analysis is of course a graph. It contains the source code elements (files, classes, methods) as nodes and the corresponding relationships between these elements as edges. Also, it has a Java API for the graph. These features make it an ideal choice for this task. Due to the amount and structure of data we are dealing with, we decided to use Neo4j[6] for data storage. It is currently the most popular open-source graph database. It also has a powerful query language called *cypher* that can be utilized for computing process metrics.

Our motivation is to provide a way of computing software process metrics quickly and easily. The main contributions of this study are the following:

- A method for automatically calculating process metrics in a graph database for files, classes and methods;
- An open-source implementation of the presented method;
- Assessment of the bug prediction capabilities of the calculated process metrics; and
- A publicly available bug database with process metrics.

The remainder of the paper is organized as follows. Section 2 presents some related work and Section 2.1 summarizes the process metrics used in other studies. In Section 3, we present the database schema that we designed, the steps of its

[1] `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`
[2] `https://github.com/about`
[3] `https://developer.github.com/v3/`
[4] `https://git-scm.com/`
[5] `https://www.sourcemeter.com/`
[6] `https://neo4j.com/`

construction and the calculation of process metrics in this database. Section 4 lists the processed software systems with some statistics concerning them. In Section 5, we present the results of machine learning algorithms applied to this database. Lastly, in Section 6 we draw some pertinent conclusions and suggest some plans for future study.

2 Related Work

Bug prediction is an intensively studied research area [12, 14, 11] and there are publicly available databases that can be used for bug prediction purposes. The biggest of these datasets is the tera-PROMISE [13] repository. It is up-to-date and it is regularly maintained. It contains, among other things, bug databases with various metrics, like rule violations, object-oriented metrics and complexity metrics. Actually, some studies utilize this dataset, but there are many researchers who build their own databases and never publish them.

D'Ambros et al. [4] made an extensive study on bug prediction. They compared the well-known bug prediction approaches. As part of their research, they created a benchmark database from several open-source projects (Eclipse, Mylyn, Lucene). This database contains bug numbers at the class level with 15 change metrics and 17 product metrics. The bug information was extracted from the commit messages and bug tracking systems by using pattern matching, as others did in earlier studies [24, 5]. They describe the whole process of building such a database, but the links to the tools used do not work anymore. They computed change metrics and bug information for files due to the file-based version control systems. In the case of Java inner classes, the same information was linked to multiple classes. Since they did not have a solution to this problem, they filtered these inner classes from the process. They found that the Weighted Churn and Linearly Decayed Entropy metrics perform the best (around 90%) for bug prediction, but the computation of these metrics is quite complex. Furthermore, they concluded that multiple metrics should be used for this purpose in order to achieve good results across multiple systems.

The Eclipse project is used quite often for studies on bug prediction. Bernstein et al. [1] used this project to examine whether temporal features are suitable for bug prediction. They gathered change information from CVS and bugs from Bugzilla and they computed several temporal features. They built non-linear models for the bug database they created and they achieved a high accuracy score (99%) on predicting defects. They concluded that temporal features (process metrics) and non-linear models are suitable for bug prediction. Moser et al. [15] also used the Eclipse project to investigate the characteristics of change metrics in bug prediction. They calculated 18 change metrics at the file level. They achieved better results with these metrics than with product metrics [16] and they showed that 3 out of 18 change metrics can achieve good results, and they are as stable as the model with all the metrics. These three metrics are the following: number of revisions, number of bug fixes, maximum size of all of its change sets.

Shihab et al. [22] also examined whether the number of the predictors can be reduced. As a data source, they used the Eclipse data set [24]. They showed that the 34 product and process metrics can effectively be reduced to 4 with very little difference in the overall prediction accuracy. They found that the most stable independent metrics were: total prior changes, number of pre-release defects and TLOC.

The study made by Krishnan et al. [10] sought to answer the research question of whether the process metrics are good bug predictors for the family of products in the evolving Eclipse product line. They replicated the results previously achieved by Moser et al. [16] and extended them with their observations. They concluded that process metrics are good bug predictors for the Eclipse product line. Furthermore, they found that a small subset of these metrics are stable and consistent across multiple projects. They are called maximum changeset, number of revisions and number of authors.

Graves et al. [6] also made a study on the bug prediction capabilities of process metrics. They computed the metrics at the module level and analyzed systems written in C. Their observation was that the best model used the weighted time damp metric and the best linear models used number of changes and age metrics. They found that the number of developers and the changeset metrics did not influence the accuracy of the fault prediction.

In an earlier paper [7], we presented a method for characterizing software bugs with product metrics. In this method, we include temporal information by building the bug database from the buggy source code elements before and after the fix, but of course more sophisticated temporal characteristics should be included.

Studies have shown that process metrics usually perform better in bug prediction than product metrics do. Rahman et al. [20] analyzed the properties of process metrics from the perspective of performance, stability, portability and stasis. They found that product metrics have a higher stasis - which means they do not change much compared to the process metrics -, thus the same elements were predicted as defective over and over. Also, product metrics are less stable and less portable across projects.

Hassan [8] went further. In his paper, he proposed complexity metrics that are based on process metrics. He analyzed 6 projects written in C and C++ and computed process metrics at the file level, but he did not give a detailed description of the method of processing and how to compute these metrics. He concluded that the proposed change complexity metrics are better fault predictors than the well-known process metrics. He also said that we should consider using these metrics instead of the simple metrics like number of prior modifications and number of prior faults.

Buginfo[7] is a tool that is used for collecting bug information from source code repositories [9]. It uses regular expressions on the commit messages to count the number of bugs in classes, as other studies did [24, 5]. It is also capable of computing process metrics, but unfortunately the tool is not maintained.

[7] `https://kenai.com/projects/buginfo`

In this study, we present a method to automatically compute the process metrics for GitHub projects. We use Neo4j, a graph database engine to store the information collected, and we utilize the cypher query language to readily compute these metrics. None of the previous studies used GitHub as a data source. Also, none of them defined a graph schema for the data they collected nor did they use a graph database. Furthermore, there are no studies to date that calculate software process metrics at the method level.

2.1 Process Metrics

In the literature, there are many software process-related metrics [17, 18, 8, 1] and they were mainly used in studies concerning bug prediction. In these studies, the authors evaluated the predictive capability of these metrics and they often compared this capability with that of product metrics [16]. They found that the age of a file and the size of the change metrics are usually better predictors than the others [15, 10]. In this study, we did not rank these metrics in any way. Here, we enumerate the definitions of the most common metrics:

- **Number of Modifications**: The number of previous modifications of the source element.
- **Number of Bug Fixes**: The number of previous modifications of the source element that reflect an intention to fix a bug.
- **Number of Versions**: The number of software versions (revisions) since the source element is created. In other words, the number of commits on the whole project since the creation of the element.
- **Number of Re-factorings**: The number of previous modifications of the source element that were committed in order to perform re-factoring.
- **Age**: The age of the source element in days, weeks or months.
- **Weighted Age**: The weighted age is calculated using the age and size of the previous modifications. [16] It may be expressed in days, weeks, or months. The formal definition is the following:

$$WeightedAge(e) = \frac{\sum_v Age(v) \times NumberOfAddedLines(e,v)}{\sum_v NumberOfAddedLines(e,v)} \quad (1)$$

  In this formula, we would like to compute the metric for the source element e. The *Age* is the age of the software version v (days, weeks, or months), where v is earlier than the version for which we want to calculate. The *NumberOfAddedLines* represents the number of lines added for source element e in version v.

- **Number of Contributors**: How many different developers contributed to the source element.

- **Number of Contributor Changes**: The number of developer changes in the code history. A developer change occurs when the next sequential modification on the same source element was performed by a different developer.
- **Sum of Added Lines**: The total sum of the lines of code added to the source element.
- **Maximum Number of Added Lines**: The maximum number of lines of code added with one commit to the source element.
- **Average Number of Added Lines**: The average number of lines of code added to the source element.
- **Number of Additions**: The number of previous commits in which new lines were added to the source element.
- **Sum of Deleted Lines**: The total sum of the lines of code deleted from the source element.
- **Maximum Number of Deleted Lines**: The maximum number of lines of code deleted with one commit from the source element.
- **Average Number of Deleted Lines**: The average number of lines of code deleted from the source element.
- **Number of Deletions**: The number of previous commits containing lines that were deleted from the source element.
- **Code Churn**: The sum of lines added minus lines deleted from the source element [17].
- **Relative Code Churn**: The normalized Code Churn metric. Normalization can be achieved with, for example, lines of code, file count or time period [18].
- **Maximum Number of Elements Modified Together**: The maximum number of distinct elements that were modified with one commit.
- **Average Number of Elements Modified Together**: The average number of distinct elements that were modified together with the source element.
- **Average Time Between Changes**: The average number of days, weeks or months that passed between consecutive modifications of the source element.
- **Author**: The identity of the original author of the source element. It may include other information about the developer, such as the total number of commits of the author and number of projects.
- **Number of Referenced Issues**: The number of distinct issues referenced in the comments of commits that contain modifications of the source element.

- **Number of Commits Without Message**: The number of previous modifications without any comment message.

Most of these metrics are based on the versioning information and the issue tracking data, but there are others, for instance, that include software management data. We did not process such data source so we will not describe them here. Furthermore, more specific characteristics may be taken into account like *Number Of Referenced High Priority Issues* if issue priority is available. Since these details may vary from system to system, we omitted these variations from the study and we concentrated on the most common ones.

Other metrics can be formed like *Change Activity Rate* [21], which is defined as the overall number of modifications relative to the age of the source element in months. This metric can be computed by a simple division. The calculation of these combined metrics is straightforward, so we will not discuss them.

Most of the metrics listed above are computed for a given version (revision), except the fixed characteristic like *Author*. In the literature, these metrics are defined for files or modules (collection of files). We defined them for source elements which may be a file, class or even a method. Furthermore, a time period can be specified for most of these metrics. For example, we can limit the interval of the calculation for the last six months. This way, we can produce metrics like *Number Of Modification In The Last Six Months*. To save space, we did not list every variation.

3 Methodology

After analyzing the available data sets (project history, static source code analysis results), we designed a graph database schema and it is shown in Figure 1. Our goal was to construct a graph with a structure that supports the computation of process metrics, so the change information should be easy to obtain. Actually, it contains seven types of nodes. The *Project* node represents the repository of a project. Since we used GitHub, it has two attributes: the GitHub user and repository identifiers. With this node, one database can be used for multiple projects. It may be useful if we are dealing with cross-repository issue referencing, which is also one of the GitHub features. In bigger companies or on GitHub, developers usually contribute to multiple repositories. If we put these repositories into this database, then the developers are connected to multiple projects, hence we can compute with this property as well. The *User* node simply represents a developer. It has one attribute, namely the number of commits. This property is provided by the GitHub API. The *Issue* node represents a bug report from the issue tracker. It has two attributes, namely *opened* and *closed*. The former is the date of the bug report, while the latter is the date of closing the bug report or it is null if it is still open.

The *Commit* node represents a software version. It has three attributes, these being *hash*, *created* and *fix*. The first one is the unique hash of the commit. The second is the time stamp of the commit's creation. *Fix* is a Boolean property and

it tells us whether this commit is a bug fix or not. A commit is treated as a fix if the commit message references a bug report. This connection is provided by the GitHub API and in the schema, it is represented as a *Referenced* edge between *Commit* and *Issue* nodes. A commit is made by one user, thus we connect the *Commit* nodes to the *User* nodes with an *Author* edge. Sometimes the commits do not have such an edge because the developer is removed from GitHub. The *Parent* edges of *Commit* nodes represent the relationship between consecutive commits. Two commits are connected if one of them is directly followed by the other. One commit may have multiple parents in the case of merge commits.

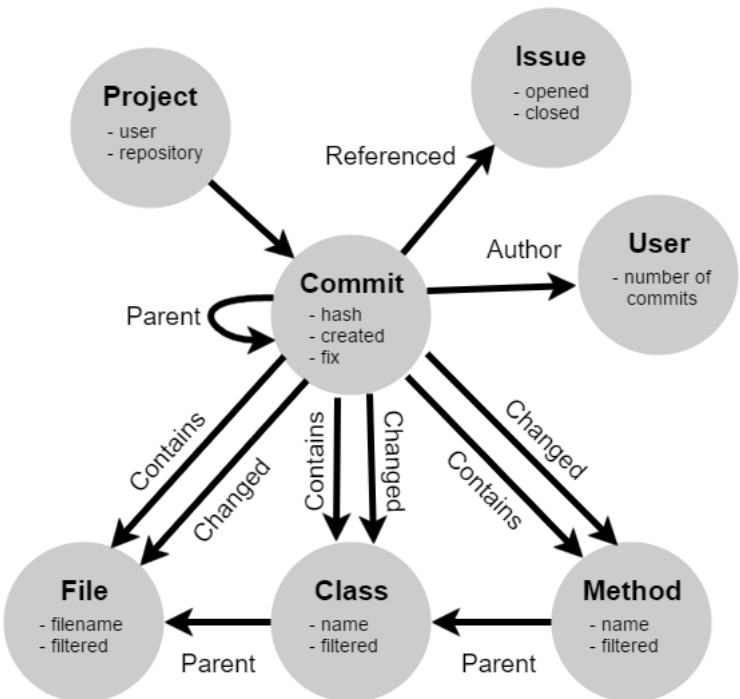


Figure 1: The Graph Database Schema

The bottom three nodes - *File*, *Class*, *Method* - represent the source code elements. The *Parent* edges between them is the containment relationship. Since we focused on the Java programming language, other containment relations are not possible. *Method* and *Class* nodes have a *name* attribute, which is the fully qualified name of the elements. The *File* nodes always have a *filename* attribute. The *filename* contains the full path of the file. In our previous study [7, 23], in order to avoid marking non-buggy test code as buggy, we filtered the test-related source code elements during the collection of bug information. This filtering is based on the file name and the qualified name. The *filtered* attribute of *File*, *Class*, and *Method* nodes indicates whether a certain file, class or method was filtered or

not. The values of *name* and *filename* attributes are unique, and this means only one node is created for a given source element that lives across multiple software versions. This way, it is easy to get the changes of a file, a class or a method. This is a crucial feature of the database in terms of creating an efficient method. The *Contains* edge between commits and source elements is responsible for showing whether a given commit actually contains the specific element.

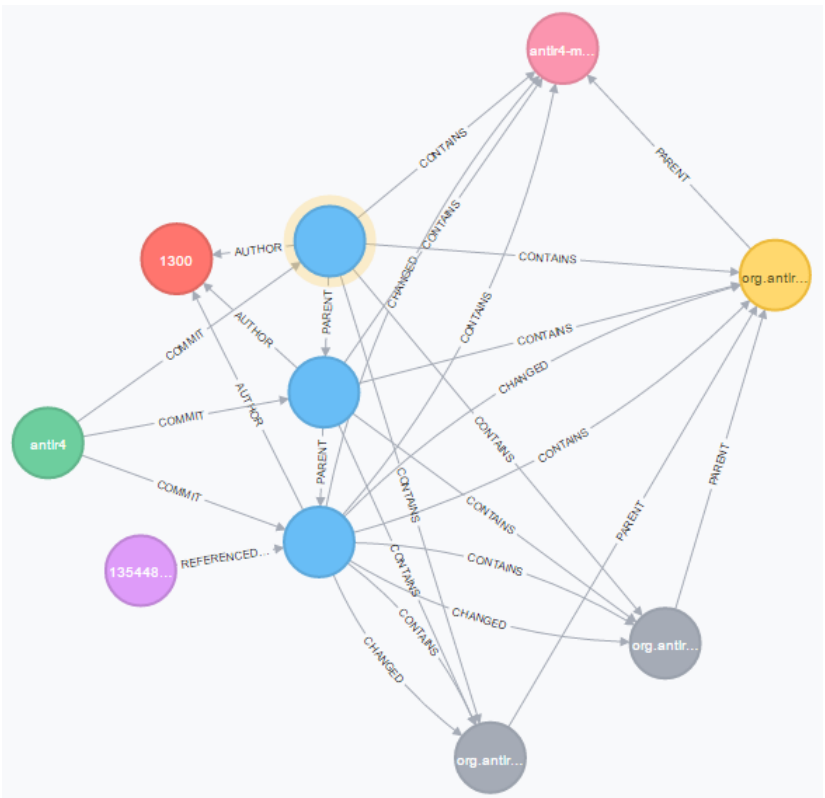


Figure 2: Example Graph (green: project, purple: issue, red: user, blue: commit, pink: file, yellow: class, grey: method)

The *Changed* edges between commits and the source elements indicate whether an element is changed during a commit. These edges have the following attributes: *added* - number of added lines, *deleted* - number of deleted lines and *modified* - number of modified lines. These values are computed from the commit patch file, which can also be got via the GitHub API. The patch file is based on files, hence for classes and methods additional mapping is required. For this task, we used the source position available from the static source code analysis. A patch file contains sections (deltas). A delta has a begin line number and an end line number. The mapping is carried out by checking whether a delta intersects the position of a source

code element. From the patch file, we can get the begin and end line numbers of a change (delta) and from the SourceMeter output, we can get the source position of a source element in a form of row numbers. Now, let us consider a delta with line values 31-46 and a method with position 24-37. The first and last three lines of a delta are unchanged, so we can subtract them from the section. After this step, we get 34-43 as line values for the delta. The intersecting range is 34-37. This means that 4 lines of the method have changed. If we look at the original version of a delta, we can extract information about the type of the change. If the size of the original is zero, then the change is an addition. Conversely, if the size of the new part is zero, then it is a deletion. Otherwise it is a modification.

Now that we have a schema definition, we can proceed to the metric calculation. Figure 2 shows a small part of the graph database created for the ANTLR4 project (more details in Section 4). In this graph, we have at least 1 of each type of nodes and relationships. As an example, let us show how to compute a simple process metric called the *Number of Modifications* in this graph. The basis of the calculation is the highlighted commit (uppermost) and the source element is the upper right method. We have to look for *Commit* nodes that are created before the subject commit and have a *Changed* relationship with the selected *Method* node. We can use the *Parent* edge between commits, or the *created* attribute for selecting the past commits. Adding the *Changed* edge to the match condition leads to the desired commits, which in this example is the lowermost *Commit* node. With a simple aggregation (counting) we get the value for the computed process metric. As we mentioned earlier, Neo4j has a query language called *cypher*. With this language, it is easy to formulate these process metrics. For example,

```
match (n:METHOD{name:'...'})<-[:CONTAINS]-(c1:COMMIT{hash:'...'}),
(n)<-[:CHANGED]-(c2:COMMIT)
where c1.created >= c2.created
return n.name as name, count(c2) as `Number of Modifications`
```

We will not go into details about the syntax of this query language. A detailed description is available on the official Neo4j website[8]. The other metrics can be formulated into a single query too, hence it is an easy way to compute them. Table 1 lists the implemented process metrics. Switching to the class level is simple, because all we need to do is change the node type in the query. To produce these metrics automatically for the selected project's selected version, we created a framework. Below, we will describe the overall picture of the framework.

An overview of the process is shown in Figure 3. The shape in the top left corner represents our data source, GitHub. The two connected elements are the first steps. These were partially described in our previous study [23]. Stated briefly, the project data is exported from the GitHub API and the source code versions are analyzed with SourceMeter during these steps. Next, the graph nodes and edges - according to the previously presented schema - are exported into CSV files. These files can be directly imported into a Neo4j graph database. The next task

[8] `https://neo4j.com/developer/cypher-query-language/`

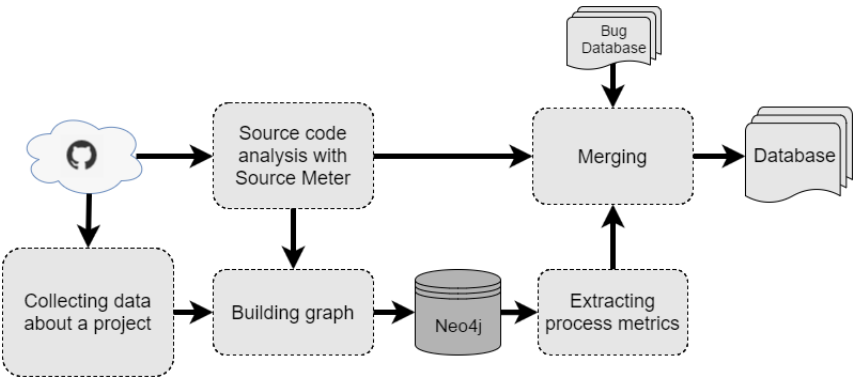


Figure 3: Overview of processes involved

is to produce the required process metrics listed in Table 1. The selection of the implemented metrics is based on the data available to us. The *cypher* queries are executed and the results are saved in separate CSV files for each metric. The next and final step of the process is to merge the available CSV files so as to produce the desired database. SourceMeter also exports the source elements into CSV files with some essential properties, like qualified name, source position and static source code metrics. In our previous article [23], we published a bug database[9] that contains bug information for files and classes. In this database, bug numbers are collected from the known and previously fixed source code defects. For a given release version, the database contains the number of reported, but not yet fixed bugs for each element. The source code elements affected by bugs are determined from bug fixing modifications. From these data sets, we created databases that contain the source code elements (files, classes, methods), the static source code metrics and the process metrics. Also, the file and class level databases include the actual number of bugs taken from our earlier studies. Here, these databases are in CSV form. The first line contains the header information - as in the previously published data sets - extended with the process metric names. The final header element is the number of bugs. The rest of the lines are the source elements taken from the original database along with the associated process metrics. The method level databases have the same structure, but since we did not produce method level bug databases previously, the number of bugs column is missing.

The tools created are published as open-source projects in the following GitHub repository:
`https://github.com/sed-szeged/BugHunterToolchain`

[9] `http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/`

Table 1: The list of implemented process metrics.

Age
Average Number of Added Lines
Average Number of Deleted Lines
Average Number of Elements Modified Together
Average Time Between Changes
Last Contributor Commits
Maximum Number of Added Lines
Maximum Number of Deleted
Maximum Number of Elements Modified Together
Number of Additions
Number of Contributor Changes
Number of Contributors
Number of Deletions
Number of Fixes
Number of Fixed in the Last Six Months
Number of Modifications
Number of Modifications in the Last Six Months
Number of Versions
Sum of Added Lines
Sum of Deleted Lines
Time Passed Since the Last Change
Weighted Age

4 Experimental Set-up

To demonstrate our method, we processed 5 of the Java projects we used in our previous study. Table 2 lists the selected projects with some basic statistics. The first column is the name of the systems, while the second column is the general domain of the projects. The next two columns contain the number of commits and the thousand lines of code. This statistics tells us that these projects are dissimilar regarding domain and size too. During the process, we had to analyze every single version of the systems to extract the change information. Although the analysis of individual versions was quick, the overall run time was quite high. For the Broadleaf Commerce project, the analysis of nearly ten thousand versions took around 200 hours. From a process perspective, it was just an initial step, and only needed to be executed once.

The next step was to build the graph databases. The total run time for all of the projects was around 6 hours. For a single version this time is negligible, thus the database can be extended efficiently with the new version. Table 3 gives statistics on the size of the graph databases. The first column shows the name of the project.

Table 2: The chosen Java projects.

Project	**Domain**	**Commits**	**kLOC**
ANTLR4	Language processing	3276	85
Broadleaf Commerce	E-commerce framework	9292	282
jUnit	Test framework	2053	36
MapDB	Database engine	1345	83
Titan	Database engine	3830	119

The next two columns are the number of nodes and the number of relationships (edges) in the graph, expressed in thousands. The next column contains the disk space occupied by the graph databases in Megabytes. The final column is the size of the results of source code analysis in Gigabytes. We can see that the graph is a compact way of storing the information about the project history and process metrics can be efficiently derived from it.

Table 3: The graph databases that we created.

Project	**kNodes**	**kEdges**	**Size of Graph (MB)**	**Size of Raw Data (GB)**
ANTLR4	24	13 069	484	18
Broadleaf Commerce	91	145 966	4 828	220
jUnit	14	7 457	300	9
MapDB	13	4 629	208	7
Titan	219	21 471	804	30

After setting up the database, we computed the process metrics for 25 release versions of the systems (5 each). The release versions were selected just like those in our previous study [23], namely at 6-monthly intervals. Due to smaller inactive periods in project development, it may happen that the bug numbers are zero in a given release version. In such cases, we dropped this release version, then the time interval between some of the versions was larger than six months.

Lastly, we constructed bug databases (at the file and class levels) for the selected 5 projects' 25 release versions with the computed process metrics. What is more, we created method-level databases - without bug information - which also include both product and process metrics.

5 Evaluation

We applied machine learning algorithms to our bug database in order to check whether it was suitable for bug prediction. In the preliminary step, similar to our previous study, we grouped the source elements into two classes based on the bug numbers. Source elements with zero bug numbers formed a non-defective class, while the others formed a defective class. The structure of the learning tables was the following: it contained a unique id for every instance; next, it contained the predictors (22 software process metrics); lastly it contained the label of the class as Boolean values (true - defective, false - non-defective). Separate learning tables were constructed for files and classes in each release version, hence we got 50 learning tables in total. The number of instances in a defective class was much smaller than the number of instances in a non-defective class. To avoid any distortion in the results, we applied random under sampling to the databases. This method helps to balance the number of positive and negative instances in the training set. To achieve more reliable results, we applied this method ten times to the data sets and computed an average.

We used Weka[10], the popular machine learning library to perform the training part. For the evaluation part, we used the same set of algorithms as in our earlier study [23]. Namely,

- NaiveBayes
- NaiveBayesMultinomial
- Logistic
- SGD
- SimpleLogistic
- SMO
- VotedPerceptron
- DecisionTable
- OneR
- PART
- J48 (C4.5)
- RandomForest
- RandomTree

[10] `http://www.cs.waikato.ac.nz/~ml/weka/`

We used 10-fold cross validation and we measured the performance with F-measure metrics that are defined by the following:

$precision = \frac{TP}{TP+FP}$

$recall = \frac{TP}{TP+FN}$

$F - measure = 2 \cdot \frac{precision \cdot recall}{precision+recall}$,

where TP(True Positive) is the number of instances that were predicted as defective and observed as defective, FP(False Positive) is the number of instances that were predicted as defective but observed as non-defective, FN(False Negative) is the number of instances that were predicted as non-defective but observed as defective.

Table 4: Average F-measure values at the class level.

Project	**#1**	**#2**	**#3**	**#4**	**#5**
ANTLR4	0.7235	0.5502	0.6771	0.7101	0.7765
Broadleaf C.	0.6772	0.6736	0.6812	0.6729	0.6923
MapDB	0.5626	0.6560	0.7179	0.7261	0.7426
jUnit	0.6989	0.6522	0.5949	0.6156	0.8127
Titan	0.5712	0.6231	0.6640	0.6543	0.7423

Table 5: Average F-measure values at the file level.

Project	**#1**	**#2**	**#3**	**#4**	**#5**
ANTLR4	0.7252	0.7548	0.6961	0.7334	0.6872
Broadleaf C.	0.6402	0.6759	0.6799	0.6969	0.6869
MapDB	0.5652	0.7879	0.6606	0.6930	0.8362
jUnit	0.7279	0.6102	0.7000	0.6792	0.5384
Titan	0.6082	0.7147	0.7108	0.7303	0.6924

The goal of our first investigation was to find out whether the created bug databases with the process metrics were suitable for bug prediction. We evaluated the 13 algorithms on all 25 release versions and only used process metrics as predictors. Our first observation was that the F-measure values at the class level generally improved with time except for the jUnit project. Table 4 shows the average F-measure values at the class level for all 25 versions. The first column contains the project names, while the next columns are the average results for each version in chronological order. The first is the earliest in time, the following is the next, and so on. From this table, we can see that the values increase slightly with time. One possible explanation for this is the nature of the metrics used. Most of the process metrics are based on temporal characteristics, hence these values

may be more reliable with bigger time intervals. At the file level, things are not so straightforward. In the case of the ANTLR4 and jUnit projects, the values do not follow this trend, as shown in Table 5. This may be due to the varying size of the training sets, but as we cannot generalize this observation, more investigation is needed to learn the reason for it.

Table 6: Comparison of F-measure values at the class level.

Project	**Product metrics**	**Process metrics**
ANTLR v4	0.7179	0.6771
Broadleaf Commerce	0.7544	0.6812
MapDB	0.6999	0.6560
jUnit	0.7233	0.6156
Titan	0.7058	0.7423

Table 7: Comparison of F-measure values at the file level.

Project	**Product metrics**	**Process metrics**
ANTLR v4	0.7061	0.6961
Broadleaf Commerce	0.6955	0.6799
MapDB	0.6306	0.7879
jUnit	0.5600	0.6792
Titan	0.5730	0.6924

In our previous study [23], as we reported F-measure values from release versions that have the most bug entries, we will now compare the results for these versions. Tables 6 and 7 allow us to compare the average F-measure values got with the two different sets of predictors for these release versions. Our previous database at the file level contains some process metrics, hence we once again applied the machine learning algorithms without these metrics. Also, we repeated the learning process at the class level. From these values, we can see that the process metrics performed worse than the product metrics at the class level in 4 out of 5 cases. At the file level, process metrics performed better than the product metrics in 3 out of 5 cases. Despite process metrics not performing well in the same cases as product metrics, in other cases the F-measure values we got indicate that the former perform more robustly than the latter. As we mentioned earlier, process metrics perform better in the later versions than in the earlier ones. At the file level, the F-measure values vary more with product metrics and the best performing algorithms are different for each version. With the process metrics, more or less the same set of algorithms are at the top for all of the cases studies. We did not include every result, due to the large amount of algorithms and learning tables. The detailed F-measure tables can be found in the online appendix.

Table 8: F-measure values at the class level.

Algorithm	**ANTLR4**	**Broadleaf**	**MapDB**	**jUnit**	**Titan**	**AVG**
RandomForest	0.7328	0.7638	0.7483	0.6702	0.7997	0.7430
DecisionTable	0.6704	0.7147	0.6719	0.6712	0.7742	0.7005
SMO	0.7059	0.7015	0.6682	0.6232	0.7788	0.6955
OneR	0.7101	0.6982	0.6065	0.6770	0.7597	0.6903
SimpleLogistic	0.6970	0.7059	0.6163	0.6345	0.7878	0.6883
PART	0.6730	0.7056	0.6755	0.6221	0.7468	0.6846
J48	0.6971	0.6953	0.6618	0.6077	0.7567	0.6837
SGD	0.6196	0.7039	0.6838	0.6125	0.7832	0.6806
RandomTree	0.6530	0.6861	0.6547	0.6365	0.7540	0.6769
Logistic	0.5827	0.6885	0.6459	0.6006	0.7441	0.6523
NaiveBayes	0.7106	0.5846	0.6868	0.6457	0.5672	0.6390
NaiveBayesMultinomial	0.7065	0.5920	0.6042	0.5349	0.6689	0.6213
VotedPerceptron	0.6432	0.6159	0.6040	0.4663	0.7290	0.6117

In Table 8, we list the F-measure values for the versions with the most bug entries. The first column contains the machine learning algorithm names that we used. The subsequent columns contain the resulting F-measure values for each project. The last column is the average F-measure value over projects. The table is ordered by the average value in decreasing order. Here, we notice that tree-, rule-, and function-based algorithms performed the best. The highest average F-measure value is 0.7430, while the lowest is 0.6117. The overall highest F-measure value in these versions is 0.7997 and it was achieved by the Titan project. The best performing algorithm is the *RandomForest* method. We should add that process metrics did not preform the best in these release versions.

Table 9 shows the resulting F-measures for the versions with the most bug entries. The structure of the table is the same as that for Table 8. The same set of algorithms perform the best in these cases as well. The performance of the Bayesian methods varies for each version, hence we cannot say that using these metrics, they are the best to predict bugs. The best results are over 0.8 (MapDB project) and the worst result is 0.5278; however, on average the F-measure values are around 0.7. The highest average F-measure value is 0.7448, while the lowest is 0.6622. In summary, the results obtained appear to indicate that the databases with the computed process metrics are suitable for bug prediction purposes and the best performing algorithms are the *RandomForest* and *DecisionTable* methods.

Next, we examined whether there were any relationships among the metrics. Since our databases contain both product and process metrics for classes and methods, we computed correlations among these values. As the results are similar to each other between the versions and listing the correlations for all 25 versions would take up too much space, we will only present the results for a single version. Instead

Table 9: F-measure values at the file level.

Algorithm	**ANTLR4**	**Broadleaf**	**MapDB**	**jUnit**	**Titan**	**AVG**
RandomForest	0.7804	0.7328	0.8180	0.6659	0.7271	0.7448
DecisionTable	0.7299	0.7152	0.8143	0.7061	0.7103	0.7352
SimpleLogistic	0.7234	0.6910	0.7869	0.7071	0.7094	0.7236
SGD	0.7033	0.7025	0.8036	0.6857	0.7159	0.7222
SMO	0.7192	0.7055	0.8085	0.6893	0.6796	0.7205
NaiveBayesMultinomial	0.6920	0.6342	0.8152	0.6732	0.7533	0.7136
OneR	0.6911	0.6642	0.7786	0.7371	0.6781	0.7098
J48	0.6553	0.6994	0.7983	0.6614	0.7304	0.7090
PART	0.6957	0.6907	0.7879	0.6552	0.6575	0.6974
RandomTree	0.6826	0.6740	0.7261	0.6933	0.6896	0.6931
VotedPerceptron	0.6267	0.6340	0.8019	0.6149	0.7406	0.6836
Logistic	0.6765	0.6950	0.6699	0.6649	0.6821	0.6777
NaiveBayes	0.6732	0.6008	0.8334	0.6757	0.5278	0.6622

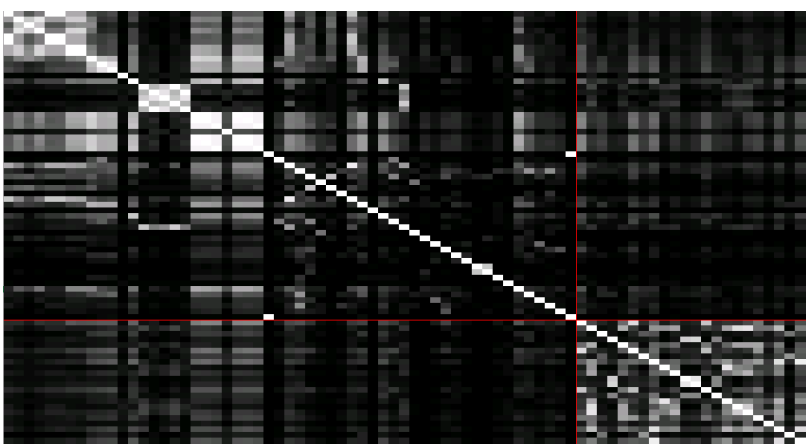


Figure 4: Correlation of method metrics

of including the correlation matrices that have over a hundred rows and columns, we illustrated these with colored tables (Figures 4, 5 and 6). The black cells denote the low absolute value of the correlation (close to zero), while the white cells denote the high absolute value of the correlation (near one or minus one). Figure 4 shows the correlation of method metrics tested. The product metrics (including rule violations) are separated from the process metrics by a red line. From this image (bottom right quarter), we can see that the process metrics correlate more with each other than with product metrics. There are some correlation values around 0.4-0.5 between a few size-based product metrics (Lines of Code, Number of State-

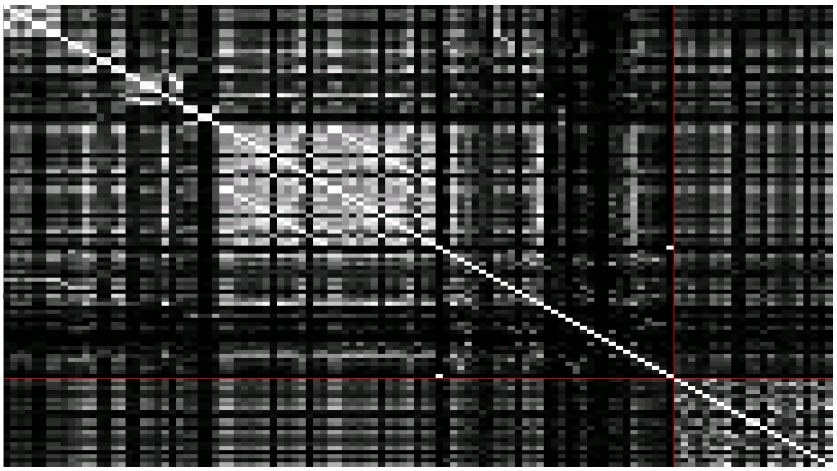


Figure 5: Correlation of class metrics

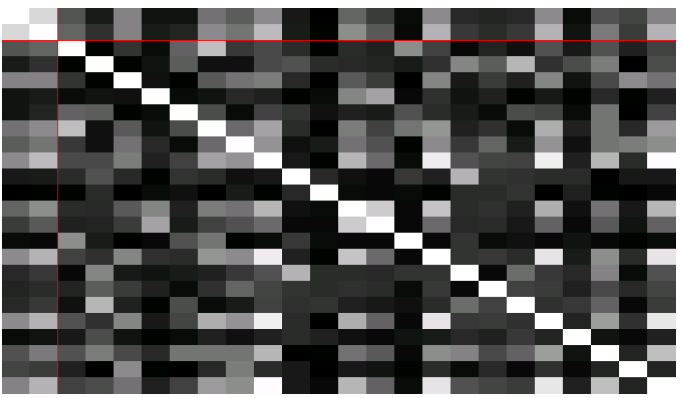


Figure 6: Correlation of file metrics

ments) and process metrics (Number of Added Lines, Number of Modifications), but in general there are no high correlation values.

If we look at the correlation results between any two class metrics in Figure 5, we notice that the relation between process and product metrics is a little clearer. Nevertheless, the correlation is still noticeably higher between the same type of metrics. At the file level (Figure 6), as the database contains only two product metrics, we cannot draw any conclusions from it.

From the correlation results presented above, we can say that process metrics are of a different nature from product metrics. These metrics characterize the source code elements from a different perspective than product metrics do. The full correlation matrices, the results of the evaluation and the databases created

are accessible as an online appendix at the following URL:
`http://www.inf.u-szeged.hu/~pgyimesi/papers/ActaCybernetica2016/`

6 Conclusions and Future Work

In this study, we presented a method that efficiently computes software process metrics in a graph database. Also, we made an implementation available on GitHub as an open-source project that is capable of computing 22 process metrics. We selected 5 Java projects and with our implementation, we processed these systems and produced databases for 25 release versions that were selected from an earlier study. The databases created contain the implemented process metrics for files, classes and methods. Afterwards, we employed our previously published bug databases (at the file and class levels) and extended them with process metrics; then we applied 13 machine learning algorithms on them to investigate whether the database was suitable for bug prediction purposes and we achieved promising results. Based on the F-measure values, we found that tree- and rule-based methods perform the best and, in particular, the *RandomForest* method performed well in every case.

In the future, we intend to implement more process metrics and experiment with new ones. We also plan to extend the list of processed systems. Furthermore, we would like to set up a method-level bug database and to evaluate the bug prediction capability of method-level process metrics.

7 Acknowledgements

I would like to express my gratitude to my supervisor Dr. Rudolf Ferenc for his useful comments and remarks. He also helped clarify certain points and issues during the study.

References

[1] Bernstein, Abraham, Ekanayake, Jayalath, and Pinzger, Martin. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[2] Bhattacharya, Pamela, Iliofotou, Marios, Neamtiu, Iulian, and Faloutsos, Michalis. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 419–429. IEEE Press, 2012.

[3] Catal, Cagatay. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[4] D'Ambros, Marco, Lanza, Michele, and Robbes, Romain. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[5] Fischer, Michael, Pinzger, Martin, and Gall, Harald. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[6] Graves, Todd L, Karr, Alan F, Marron, James S, and Siy, Harvey. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[7] Gyimesi, Péter, Gyimesi, Gábor, Tóth, Zoltán, and Ferenc, Rudolf. Characterization of source code defects by data mining conducted on GitHub. In *International Conference on Computational Science and Its Applications*, pages 47–62. Springer, 2015.

[8] Hassan, Ahmed E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[9] Jureczko, Marian and Madeyski, Lech. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[10] Krishnan, Sandeep, Strasburg, Chris, Lutz, Robyn R, and Goševa-Popstojanova, Katerina. Are change metrics good predictors for an evolving software product line? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 7. ACM, 2011.

[11] Madeyski, Lech and Jureczko, Marian. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[12] Malhotra, Ruchika. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[13] Menzies, T., Krishna, R., and Pryor, D. The Promise Repository of Empirical Software Engineering Data, 2015. http://openscience.us/repo. North Carolina State University, Department of Computer Science.

[14] Menzies, Tim, Milton, Zach, Turhan, Burak, Cukic, Bojan, Jiang, Yue, and Bener, Ayşe. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[15] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 309–311. ACM, 2008.

[16] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190. IEEE, 2008.

[17] Munson, John C and Elbaum, Sebastian G. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[18] Nagappan, Nachiappan and Ball, Thomas. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292. IEEE, 2005.

[19] Radjenović, Danijel, Heričko, Marjan, Torkar, Richard, and Živkovič, Aleš. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[20] Rahman, Foyzur and Devanbu, Premkumar. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[21] Ratzinger, Jacek, Pinzger, Martin, and Gall, Harald. EQ-Mine: Predicting short-term defects for software evolution. In *International Conference on Fundamental Approaches to Software Engineering*, pages 12–26. Springer, 2007.

[22] Shihab, Emad, Jiang, Zhen Ming, Ibrahim, Walid M, Adams, Bram, and Hassan, Ahmed E. Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[23] Tóth, Zoltán, Gyimesi, Péter, and Ferenc, Rudolf. A Public Bug Database of GitHub Projects and Its Application in Bug Prediction. In *International Conference on Computational Science and Its Applications*, pages 625–638. Springer, 2016.

[24] Zimmermann, Thomas, Premraj, Rahul, and Zeller, Andreas. Predicting defects for Eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.

Acta Cybernetica **23** (2017) 561–572.

The Structure of Pairing Strategies for k-in-a-row Type Games

Lajos Győrffy[a], András London[b], and Géza Makay[c]

Abstract

In Maker-Breaker positional games two players, Maker and Breaker, play on a finite or infinite board with the goal of claiming or preventing the opponent from getting a finite winning set, respectively. For different games there are several winning strategies for Maker or Breaker. One class of winning strategies is the so-called pairing (paving) strategies. Here, we describe all possible pairing strategies for the 9-in-a-row game. Furthermore, we define a graph of the pairings, containing 194,543 vertices and 532,107 edges, in order to give them a structure them. A complete characterization of the graph is also given.

Keywords: positional games, pairing strategies, hypergraphs

1 Introduction

The positional game k-in-a-row is played by two players on an infinite (chess)board. In the usual version of the game, the players alternately place their own marks (crosses or cirles) on previously unmarked squares, and whoever gets a *winning set* (k-consecutive squares horizontally, vertically or diagonally) first with his own marks, wins.

In the Maker-Breaker (M-B) version of the game, Maker is the one who tries to occupy a winning set, while Breaker tries to prevent Maker from winning. There is a connection between the normal (Maker-Maker) and the M-B versions of a game. If the first player wins the normal game, she wins the M-B one as well. If Breaker wins the M-B game, then the second player can obtain a draw in a normal game. However, the converse statements are not true (see, for instance, the Tic-Tac-Toe game). The M-B version is easier for Maker because she does not need to frustrate Breaker's moves. For a comprehensive book on Maker-Breaker positional games, see e. g. Beck [2]. For different values of k, there are several winning strategies either for Maker or for Breaker. One group of these are the *pairing (paving) strategies*.

[a]University of Szeged, Bolyai Institute, E-mail: lgyorffy@math.u-szeged.hu
[b]University of Szeged, Institute of Informatics, E-mail: london@inf.u-szeged.hu
[c]University of Szeged, Bolyai Institute, E-mail: makayg@math.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.8

A pairing strategy generally means that the possible moves of a game are paired up; if one player plays one, the other player plays its pair (see [7]). A winning pairing strategy of Breaker in the k-in-a-row type games is a pairing of the squares of the board such that each winning set contains at least one pair. Using the above-mentioned pairing strategy, Breaker takes at least one square from each winning set, and wins the game. If there is a winning pairing strategy for Breaker in a game, then we say that there is a *good pairing for the game*. It was shown in [2, 3] (and it is illustrated in Figure 1) that there is a good pairing for the k-in-a-row game, if $k \geq 9$.

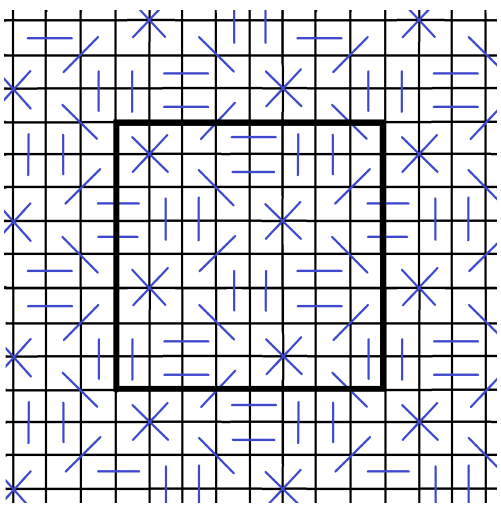


Figure 1: Hales-Jewett pairing blocks the 9-in-a-row

However, the best bound for Breaker's win was given by A. Brouwer, under the pseudonym T.G.L. Zetters [5]. He showed that Breaker will win the 8-in-a-row by tiling the board into smaller sub-boards. The best bound for Maker's win was given by Allis et al. [1]. They showed that the first player wins the classical version of the game on the 15×15 table, which implies that Maker (following first player's winning strategy from the normal game on the 15×15 table) wins the 5-in-a-row M-B game on the infinite board, but the normal (Maker-Maker) game is still open there. The cases $k = 6, 7$ are also open.

In the case of $k = 9$ —which is the main subject of this paper—there are only highly symmetric (8- or 16-toric) good *domino pairings* of the board (see Győrffy et al. [6]). An 8-toric pairing means that the pairing is an extension of an 8×8 square, while domino pairing means that all pairs consist of only neighboring (vertically, horizontally or diagonally) squares called *dominoes*. We know from [6] that all 16-toric good pairings can be deduced from two (or more) different 8-toric pairings.

After discussing the preliminaries, in Section 3 we will define a related game, namely the 8×8 torus game. An 8-toric good pairing for the 9-in-a-row is also a good pairing for the 8×8 torus game, but the reverse is not true. In Section 4,

we introduce a program that lists all (194543) 8-toric good pairings. In Section 5, we reveal a connection between the pairings, namely two pairings are neighboring if we can obtain them from each other by a natural method. With this connection, we create a graph on the pairings. In Section 6, we analyze the network of pairings, which turns out to be a huge and sparse graph (194543 nodes and 532107 edges).

After, we investigate another similar game, namely the k-in-a-row game on the hexagonal board. For this game, Breaker has winning pairing strategies if $k \geq 7$. For $k = 7$ we can list the 26 different 6-toric pairings which give us a similar but smaller graph of pairings than before.

2 Preliminaries

2.1 Pairing strategies

Given a hypergraph $\mathcal{H} = (V, E)$, where $V = V(\mathcal{H})$ and $E = E(\mathcal{H}) \subseteq \mathcal{P}(\mathcal{H}) = \{S : S \subseteq V\}$ are the set of vertices and edges, respectively. A bijection $\rho : X \to Y$, where $X, Y \subset V(\mathcal{H})$ and $X \cap Y = \emptyset$ is a **pairing** on the hypergraph $\mathcal{H}$. An $(x, \rho(x))$ pair **blocks** an $A \in E(\mathcal{H})$ edge, if A contains both elements of the pair. If the pairs of ρ block all edges, we say that ρ is a **good pairing** of $\mathcal{H}$.

Pairings are one way of showing that Breaker has a winning strategy in positional games. A good pairing ρ for a hypergraph $\mathcal{H}$ can be turned into a winning strategy for Breaker in the M-B game on $\mathcal{H}$: following ρ on $\mathcal{H}$ in a M-B game, for every $x \in X$ element chosen by Maker, Breaker chooses $\rho(x)$ or in the case of $x \in Y$ vice versa (if $x \notin X \cup Y$, then Breaker can choose an arbitrary vertex). Hence Breaker can block all edges and win the game.

Since in our paper k-in-a-row type games play an important role, we will define $\mathcal{H}_k$, the hypergraph of the k-in-a-row games.

Definition 1. *The vertices of the* ***k-in-a-row hypergraph*** $\mathcal{H}_k$ *are the squares of an infinite (chess)board, i. e., an infinite square grid. The edges of the hypergraph* $\mathcal{H}_k$ *are the* k*-element sets of consecutive squares in a row horizontally, vertically or diagonally. We shall refer to the infinite rows as* lines.

In the previous section (see Figure 1) we saw the first example of a Breaker winning pairing strategy i. e., a good pairing for $\mathcal{H}_9$. A counting type proposition indicated that there is no good pairing strategy for $\mathcal{H}_k$, if $k < 9$. We will use this well-known proposition; hence we formulate and prove the exact statement here.

For a hypergraph $\mathcal{H}$ let $d_2(\mathcal{H})$ (briefly d_2) be the greatest number of edges that can be blocked by two vertices of $\mathcal{H}$, i. e., d_2 is the maximal co-degree of $\mathcal{H}$.

Proposition 1. *If there is a good pairing* ρ *for the hypergraph* $\mathcal{H} = (V, E)$*, then* $d_2|X|/2 \geq |\mathcal{G}|$ *must hold for all* $X \subset V$*, where* $\mathcal{G} = \{A : A \in E, A \subset X\}$.

Proof. [4] We will refer to X as a sub-board. The edges of $\mathcal{G}$ can be blocked only by pairs coming from X. There are at most $|X|/2$ such pairs of ρ on the sub-board of size $|X|$. A pair blocks at most d_2 edges, while $|X|/2$ pairs block at most $d_2|X|/2$. So, if there are more edges on the sub-board, there can not be a good pairing. □

With the help of Proposition 1, we may conclude that there is no pairing strategy for $\mathcal{H}_k$ if $k < 9$. In the hypergraph $\mathcal{H}_k$, $d_2 = k - 1$ because a pair blocks at most $k - 1$ edges and this happens if and only if the pair is a domino. If X is an $n \times n$ sub-board for sufficiently large n, then $|\mathcal{G}| = 4n^2 + O(n)$ because four edges start from a square (a vertical, a horizontal and two diagonal, except at the border). By Proposition 1, we get $(k-1)n^2/2 \geq 4n^2 + O(n)$; that is, $k \geq 9 + O(1/n)$. One can even compute $O(n)$ exactly: $O(n) = -6(k-1)n + 2(k-1)^2$.

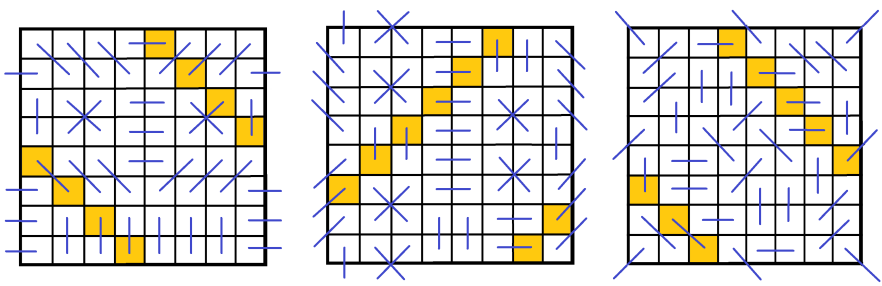


Figure 2: Some good pairings for the 9-in-a-row and some diagonal torus lines

For $\mathcal{H}_9$ there are some other good pairings besides the Hales-Jewett pairing, three of which can be seen in Figure 2. One can check that their extensions – by just repeating the 8×8 squares on the infinite grid – are good pairings for $\mathcal{H}_9$.

2.2 Conditions for good pairings of $\mathcal{H}_9$

Consider an $n \times n$ square sub-board of the infinite board. Proposition 1 tells us that $(k-1)n^2/2 \geq 4n^2 + O(n)$, which implies that $k \geq 9 + O(1/n)$. This suggests that one must use the pairs "optimally" to block $\mathcal{H}_9$; that is, a pair should block the maximum possible edges of $\mathcal{H}_9$. We will now define the notion of optimality more precisely as follows.

Definition 2. *A pairing is optimal if*

1. *Every pair blocks exactly $k - 1$ edges.*
2. *There are no overblockings, i. e., every edge is blocked by exactly one pair.*
3. *There is no empty square, i. e., every square is contained in a pair.*

Observation 1. *Let us consider an optimal good pairing for $\mathcal{H}_9$. Then this pairing is a domino pairing in which the dominoes follow each other by 8-periodicity in each line and all squares are covered by a pair.*

Proof. The first condition in Definition 2 implies that the pairing is a domino pairing, while the second gives the 8-periodicity since otherwise it would cause either overblocking or result in an unblocked edge. The lack of empty squares is just a repetition of the third condition. □

Definition 3. *We call a square of a pairing* **anomaly** *in the cases where either the 8-periodicity is violated or a non-domino pair appears or there is an empty square in the pairing.*

The above-mentioned pairings are anomaly-free pairings. However, according to Proposition 1 there might be $O(n)$ anomalies even in a good pairing of $\mathcal{H}_9$. But this conjecture was shown to be false (see Section 6 of [6]), so all good pairings for $\mathcal{H}_9$ are anomaly-free.

Definition 4. *A pairing of the infinite board is* ***k-toric*** *if it is a repetition of a $k \times k$ square, where k is the smallest possible value.*

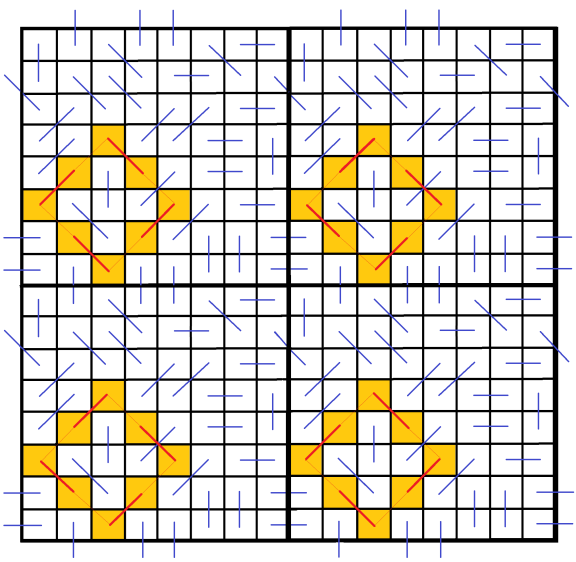


Figure 3: A 16-toric good pairing

Theorem 6 in [6] states that all good pairings for $\mathcal{H}_9$ are either 8-toric or 16-toric. The Hales-Jewett pairing in Figure 1 and the three pairings in Figure 2 are all 8-toric. We can see a 16-toric (but not 8-toric) good pairing in Figure 3. The four 8×8 squares differ from each other only in the colored squares, where bold pairs denote the actual pairs and thin ones denote the difference. Note that considering the bottom-left (which is the same as the upper-right) or the bottom-right (upper-left) 8×8 squares separately in Figure 3 we get two different good 8-toric pairings. In [6], Corollary 8 states that all 16-toric good pairings of $\mathcal{H}_9$ are just a combination of two (or more) different 8-toric ones.

In the remaining part of the study, we will describe the 8-toric good pairings for $\mathcal{H}_9$.

3 Torus game

Let us now commence with by a new M-B game that is closely related to the 9-in-a-row game.

Definition 5. *The 8×8 Maker-Breaker torus game is played on the 64 squares of the discrete torus, where there are 32 winning sets, namely the eight rows and columns and the diagonal torus lines of slope ± 1. We will call $\mathcal{T}_8$ the hypergraph of that game.*

In Figure 2, we can see some diagonal torus lines marked in yellow.

Remark 1. In the case of $\mathcal{T}_8$, there are 32 winning sets and 64 squares on which there may be 32 pairs. Hence, if we consider the four directions (horizontal, vertical and two diagonals), then the case of $k = 8$ is the unique value of k for which there may be good pairings in such a way that all winning sets contain exactly one pair.

Observation 2. *An arbitrary good 8-toric pairing for $\mathcal{H}_9$ provides a good pairing for $\mathcal{T}_8$.*

Proof. To identify the squares of an 8×8 torus, we fix one square of the torus and call it $a1$, and then we use the usual chess notation. Choose a good 8-toric pairing of $\mathcal{H}_9$ and cut an 8×8 square from that. Due to the 8-toric condition, a 8-long line either contains a domino pair (e. g. $e4 - e5$) or there are two half pairs in the square (e. g. $e1$ and $e8$), which are neighbors on the torus. □

Remark 2. The reverse is not true, because the 8×8 torus has good pairings which are not domino types. What is more, there are (non-domino) pairs which can block more than one edge (e. g. $a4 - d8$ blocks a slope $+1$ and a -1 line), which allows us unpaired squares. However, by choosing only domino pairs we can always extend a good pairing of the torus to an 8-toric pairing of $\mathcal{H}_9$.

Remark 3. There are (only 16-toric) pairings which are good for $\mathcal{H}_9$, but on an 8×8 section of it the given pairing is not necessarily good for $\mathcal{T}_8$ (e. g. the central 8×8 section in Figure 3). There are some (non-domino) pairings which are good for $\mathcal{T}_8$, but their extensions are not good for $\mathcal{H}_9$. And all 8-toric, domino pairings good for $\mathcal{H}_9$ are also good domino pairings for $\mathcal{T}_8$ as well, and vice versa.

To find all the good domino pairings for the 8×8 torus is a finite task, which is not hard using a computer. However, one has to check the torus symmetries to list the non-isomorphic pairings, which makes solving the problem difficult. Next, we will investigate the difficulties and present our searching program.

4 Generating the possible pairings on an 8×8 board

The goal was to write a program that can find all possible pairing strategies of the infinite board for the 9-in-a-row game. Since these strategies may be constructed

from 8- or 16-toric pairings and both the 8- and the 16-toric pairings may be deduced from pairings on an 8×8 section of the board, we looked at the 8×8 pairings only (with the toric expansion in mind).

We store a pairing in an 8×8 table, each cell representing the actual pair of the cell according to the 8 possible pairs: 0 means East, 1 South-East, etc., 7 North-East. Naturally, if a cell's pair is on the Eastside, then its pair has its own pair on the Westside, i. e., we fill the table two cells at a time. The algorithm itself is the usual backtracking algorithm, namely we find possible pairs for the next cell in the table having no pair so far, try all those by recursively calling the table filling function. While checking whether a pair is possible, we also make sure that there can be no overblocking, so we keep track of the blocked edges. A detailed example can be found on a webpage [8].

From previous experience, we know that speed is crucial. There are too many such pairings, so we try to reduce the number of cases. We consider two pairing strategies on the infinite board the same, if they can be transformed into each other by a translation, mirroring and rotation. So, in order not to find the same pairing several times, we apply all transformations for any pairing found on the 8×8 table. From these transformed pairings we select the smallest in the lexicographical order. This also means that such a pairing must start with 0 and 4 in the first row of the 8×8 table, so we can also reduce the number of cases by starting the table with these two numbers.

Naturally, we keep in mind the fact that the 8×8 table is expanded in an 8-toric way to the whole infinite board, while applying these transformations. More precisely:

1. We either mirror or not (2 possible cases) the table about the vertical line between columns 4 and 5.

2. We rotate the table by 0, 90, 180, and 270 degrees (4 possible cases).

3. We try all toric (that is, modulo 8) translations, which results in a table starting with a 0-4 pair.

We select the lexicographycally smallest table as a representative for the actual pairing.

The method above reduced the number of all pairing checked to 6,210,560 and the program found the 194,543 different pairings in about 4 minutes on a desktop computer with a 3.2 GHz Core i7 processor using 12 Mb of memory. The pairings themselves can be downloaded from a webpage [8]. Interestingly, the number of different pairings turns out to be a prime number.

Since we have many such different pairings, an obvious way of finding a structure is to store the pairings in a graph. Next, we will present a natural method for finding connections between pairings.

5 Creating a graph of "connected pairings"

When trying to find pairings by hand one may observe that one can move a pair along the blocked edge by one step to create a new pairing using the following method:

1. Move the first pair on the table. This move creates a cell (say A) without a pair, and another cell (say B) with two pairs.
2. Move the pair containing cell B which was not the previously moved pair so that cell B has one pair after the move. But then another cell may have two pairs.
3. Repeat step 2 as long as it creates a cell with two pairs.
4. This method will terminate when the last move creates a new pair for cell A, which had no pair before the move.

Naturally, we must not forget that we are on an 8-toric pairing and we must move the pairs accordingly. Since we are on a finite table, this method will either end at step 4, or create a repeating cycle. But the latter case is not possible. Note that cell A cannot be part of the cycle, as it has no pair, and it would break the repetition. The first move that is entered in the cycle creates a hole "behind" (outside the cycle), and when the cycle comes to the same cell, the pair will move backwards, and the cycle does not begin again.

Also, it is not hard to see that we get an optimal pairing with this method. Since the original pairing was optimal, moving a pair (in an 8-toric way) along the blocked edge keeps that direction blocked. Since the method above ends in step 4, there are no cells without a pair. We also move the pairs on a torus, so no overblocking is possible.

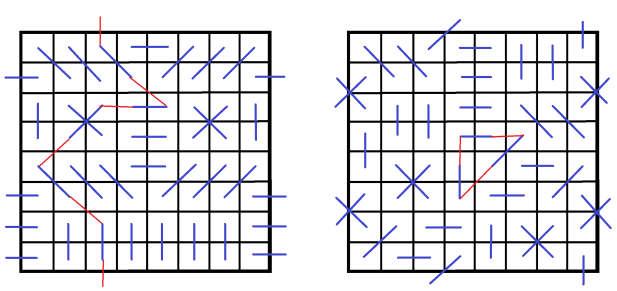


Figure 4: Two examples of connections between pairings

We say that two pairings are connected, if one can obtain the second pairing from the first one via this method (of course, we consider only different pairings as defined in the previous section). This relation is symmetric, meaning that moving

back the last pair of the above method brings us back to the first pairing from the second one. This creates a graph where the vertices are the pairings and the edges are defined by this moving transition. Figure 4 shows two examples for this moving transition. In both cases, the first pairing contains only blue pairs, and the red dominoes show the transition to the other pairing. After computing all possible different pairings, our program readily constructs this graph. It tries to move all pairings (by trying to free up each cell in the 8×8 board), and uses the method described in the previous section to find the lexicographycally smallest representative for the new pairing. It takes about 1 minute to finish this task on the same hardware as that described above.

In the next section, we will investigate the properties of the graph we obtained.

6 Analyzing the graph

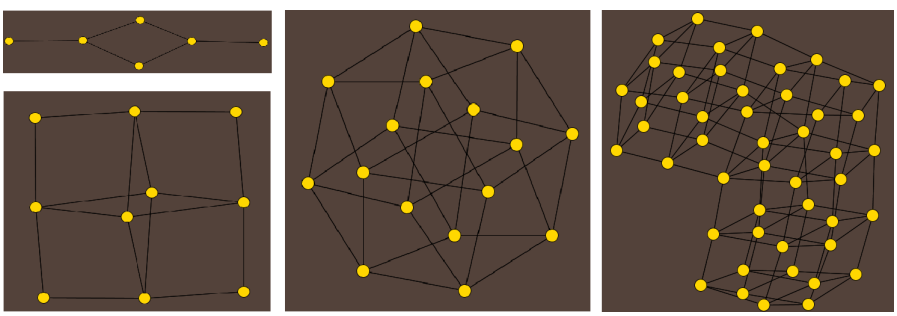


Figure 5: Some components of the given graph

Table 1: Basic parameters of the constructed graph

vertices	edges	#components	max degree	min degree	avg. degree
194543	532107	14	11	1	5.47

Table 2: Degree distribution of the graph

1	2	3	4	5	6	7	8	9	10	11
17	392	395	39811	66185	53222	25309	7547	1472	183	10

The basic parameters of the above graph can be seen in Table 1. It has 194,543 vertices and 532,107 edges. The graph is not connected, which means that we cannot reach an arbitrary pairing from another by repeating the moving transition described previously. One of the 14 components of the graph is a *giant component* containing almost all the vertices (194,333). The diameter of this component is 34, which tells us that even this giant component does not seem to be a "small-world" network. There are 5-5 smaller components of 10 and 16 vertices and 1-1

components of size 6, 26, 48. Note that the components containing 16 vertices are the net of a 4-dimensional cube. In Figure 5, there are some small components.

The graph is triangle-free and the length of all induced cycles is four. The degree distribution of the graph can be seen in Table 2 above. The average degree is 5.47, meaning that there are 5.47 transitions on average from a pairing to other pairings. Two examples for 1-degree vertices are in the smallest component of the graph pictured on the top left of Figure 5. We will show this component in detail.

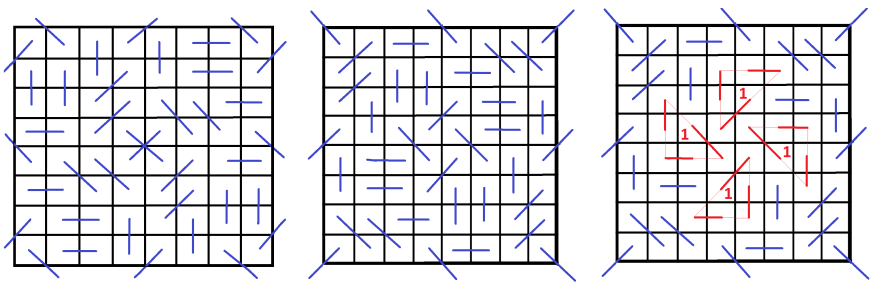


Figure 6: The same pairing twice and its four cycles

In Figure 6, we can see the same pairings on the left and middle (indicating that identifying two identical pairings is not always easy). We can see that this pairing has a rotational symmetry as well. On the right hand side of Figure 6, we can see the same pairing with its four 3-long cycles marked in red, whose cycles have two cases. Either there is a domino right to the right angle of the triangle (shown in bold in the picture, state 1) or left to the right angle (represented by thin dominoes, state 2). Note that if we change one or more cycles from state 1 to state 2 or vice versa, then we obtain another (different) good pairing. Furthermore, if we apply the moving transition described above to an arbitrary domino (not only in the cycles), then only the three dominoes in exactly one of the four cycles change.

Because of the rotational symmetry, there are six states of that group of pairings: All cycles are in state 1; one in state 2 the others in state 1; two adjacent ones are in state 1 and the other two in state 2; two non-adjacent dominoes are in state 1 and the others in state2; three are in state 2 and all four are in state 2.

It is not hard to check that the small component with six vertices is produced by these six (slightly) different pairings.

7 The case of the hexagonal board

Now we will examine the k-in-a-row Maker-Breaker game on a hexagonal board, which is the same as on a square grid with only one diagonal direction. Then Proposition 1 states that there cannot be pairing strategies if $k \leq 7$. For $k = 7$ there are good pairings, as the next theorem states. All pairings are 6-toric, which means that selecting the 6×6 torus game on a square grid with three directions

(vertical, horizontal and slope -1 diagonal) Breaker can apply the good pairings of the 7-in-a-row in this game as well.

Theorem 3. *There exists a good pairing of the 7-in-a-row game on a hexagonal board or on a square grid with three directions (vertical, horizontal and e. g. slope -1 diagonal).*

Proof. One can check that the pairing in Figure 7 is a good pairing. □

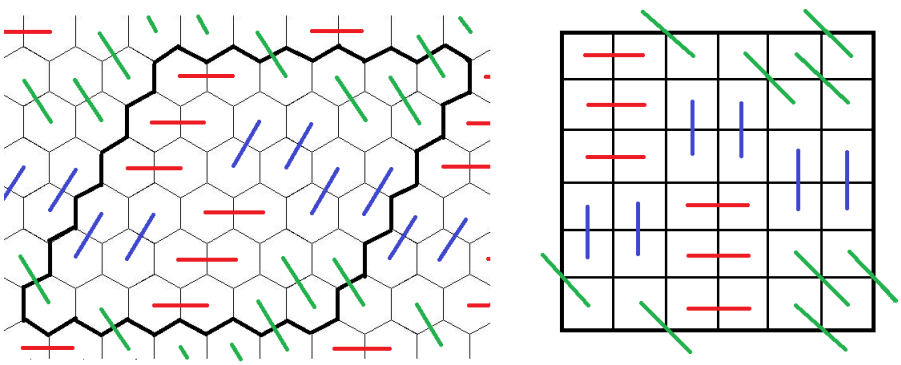


Figure 7: A good pairing for the 7-in-a-row

Using a modified version of the earlier program, we find that there exists 26 different good pairings for the 7-in-a-row game on a hexagonal board. The symmetries are different from the case of a rectangular board, because here the three directions are equivalent. Hence, there are only 6-toric pairings and there are three rotational symmetries $(0, 120, 240)$ and three reflectional symmetries on the hexagonal grid.

In Figure 8, we can see the graph got in a similar way as that in Section 6. Because of the symmetries, it contains triangles. The graph has 26 vertices and 53 edges. The average degree is 4.08 and the minimal and maximal degrees are two and six. The graph is connected and its diameter is seven.

8 Summary

In this study, we investigated the 9-in-a-row Maker-Breaker positional game and focused on pairing strategies which guarantee that Breaker wins. We found all of the different 8-toric pairing strategies using a computer program. The main concepts of the program were also described. In order to find a structure of the 194,543 pairings, we arranged them into a graph where the vertices are the pairings themselves and the edges are the moving transitions of pairs. After our analysis of this graph, we turned to the hexagonal 7-in-a-row game and provided a smaller (instead of 194,543 only 26 vertices) and more transparent example for the pairing strategies of the k-in-a-row type games.

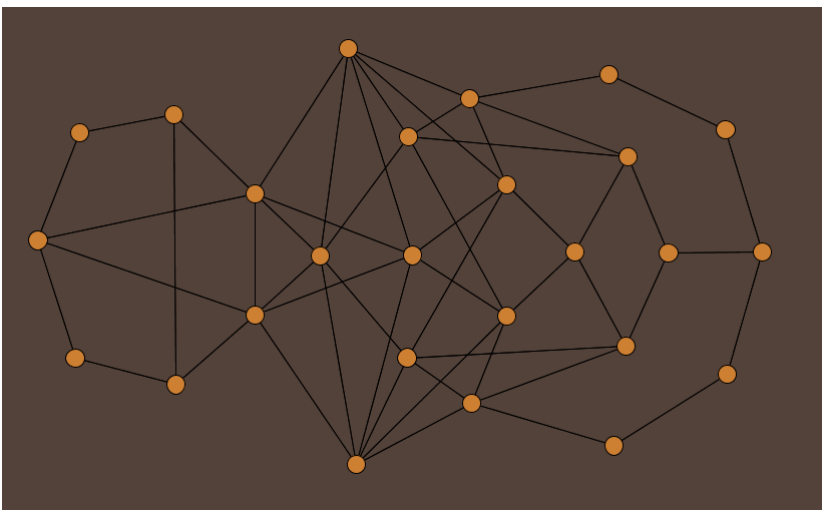


Figure 8: The graph of the 7-in-a-row game on a hexagonal board.

References

[1] L. V. Allis, H. J. van den Herik and M. P. Huntjens. Go-Moku solved by new search techniques. *Proc. 1993 AAAI Fall Symp. on Games: Planning and Learning, AAAI Press Tech. Report FS93-02*, pp. 1-9, Menlo Park, CA.

[2] J. Beck. *Combinatorial Games, Tic-Tac-Toe Theory.* Cambridge University Press, 2008.

[3] E. R. Berlekamp, J. H. Conway and R. K. Guy. *Winning Ways for your mathematical plays*, Volume **3**, 2nd Edition. A. K. Peters, MA, 2003.

[4] A. Csernenszky, R. Martin and A. Pluhár. On the Complexity of Chooser-Picker Positional Games. *Integers* 11, 2011.

[5] R. K. Guy and J. L. Selfridge, Problem S.10, *Amer. Math. Monthly* **86** (1979); solution T.G.L. Zetters **87** (1980) 575–576.

[6] L. Győrffy, G. Makay and A. Pluhár. Pairing strategies for the 9-in-a-row game. Submitted, 2016.
`http://www.math.u-szeged.hu/~lgyorffy/predok/9_pairings.pdf`
downloaded: 01. 09. 2016.

[7] A. W. Hales and R. I. Jewett. Regularity and positional games. *Trans. Amer. Math. Soc.* **106** (1963) 222–229; M.R. # 1265.

[8] G. Makay. Personal homepage
`http://www.math.u-szeged.hu/~makay/amoba/` downloaded: 06. 04. 2016.

Acta Cybernetica **23** (2017) 573–597.

The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors

István Kádár[a]

Abstract

In a software system, most of the runtime failures may come to light only during test execution, and this may have a very high cost.

To help address this problem, a symbolic execution engine called RTE-Hunter, which has been developed at the Department of Software Engineering at the University of Szeged, is able to detect runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without actually running the program in a real-life environment.

Applying the theory of symbolic execution, RTEHunter builds a tree, called a symbolic execution tree, composed of all the possible execution paths of the program. RTEHunter detects runtime issues by traversing the symbolic execution tree and if a certain condition is fulfilled the engine reports an issue.

However, as the number of execution paths increases exponentially with the number of branching points, the exploration of the whole symbolic execution tree becomes impossible in practice. To overcome this problem, different kinds of constraints can be set up over the tree. E.g. the number of symbolic states, the depth of the execution tree, or the time consumption could be restricted.

Our goal in this study is to find the optimal parametrization of RTEHunter in terms of the maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to find more runtime issues in a shorter time.

Results on three open-source Java systems demonstrate that more runtime issues can be detected in the 0 to 60 basic block-depth levels than in deeper ones within the same time frame. We also developed two novel search strategies for traversing the tree based on the number of null pointer references in the program and on linear regression that performs better than the default depth-first search strategy.

[a]University of Szeged, Department of Software Engineering Árpád tér 2. H-6720 Szeged, Hungary, E-mail: ikadar@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.9

1 Introduction

Nowadays, in software engineering it is a big challenge to produce huge, reliable and robust software systems. About 40% of the total development costs go on testing [29]; and the maintenance activities, especially the bug fixing of the system also require a considerable amount of resources [35]. In this area, symbolic execution has proven to be a practical technique for building automated test case generation and bug finding tools [6, 7, 15, 34], which supports the maintenance phase of the software engineering lifecycle.

In the context of software testing, the key goal of symbolic execution is to explore as many different program paths as possible in a given amount of time, and for each path to (1) generate a set of concrete test input values which exercises that path during a normal execution, and (2) check for the presence of various kinds of errors including uncaught exceptions, memory corruption and security vulnerabilities.

Our symbolic execution engine called RTEHunter developed at the Department of Software Engineering at the University of Szeged was developed with the goal of detecting runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without actually running the program in a real-life environment.

Symbolic execution [21] is based on the notion that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, hence a symbolic state is a set of concrete program states. When the execution of the program arrives at a branching condition containing a symbolic variable which has an unknown value, the condition cannot be evaluated and the execution continues on both branches. The execution paths created this way compose a tree called the symbolic execution tree.

However, as the number of execution paths increases exponentially with the number of branching points these tools struggle to achieve scalability. To overcome this problem the symbolic execution engines set up different kinds of constraints over the tree. For example, the number of symbolic states, the depth of the execution tree, or the runtime is limited. With RTEHunter, the maximum depth of the symbolic execution tree (i.e. the symbolic state depth) and the maximum number of states can be adjusted and arbitrary strategies of the tree traversal can also be implemented.

The main aim of this study is to find the optimal parametrization of RTEHunter in terms of the maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to identify more runtime issues in less time. This means we have to figure out which part of the whole execution tree contains most of the runtime issues while taking into consideration the time consumption of the exploration. Moreover, the search strategy is also essential to direct the exploration towards those states in the sub-tree where it is more likely to find issues and skip those that are supposed to be error-free. The maximum depth limits the height of the tree, and with a fixed depth the maximum number of states defines its width, while the search strategy determines the order in which the states in this

limited size tree should be traversed.

The main contributions of this study are the following:

- We discovered how the maximum number of states affects the execution time and the number of errors found without any constraint on the depth.
- We learned how the maximum number of states together with the maximum depth of the symbolic execution tree affects both the amount of runtime issues detected and the analysis time required.
- As our main contribution, we propose two novel search strategies that successfully detect more runtime issues by guiding the search towards the more error-prone parts of the source-code.

2 Background

2.1 Overview of Symbolic Execution

During execution, each program performs operations on the input data in a predefined order. The main idea behind symbolic execution [21] is to use symbolic variables instead of the actual data as input values and represent the values propagated during execution as symbolic expressions. A symbolic variable is a set of the possible values of a concrete variable in the program, hence a symbolic state is a set of concrete states. The output values computed by the program are then expressed as a function of the input symbolic variables.

When the execution encounters a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable whose value is unknown or uncertain, the expression cannot be evaluated, implying that the execution continues on both branches accordingly. This way all of the possible execution branches of the program can be simulated in theory.

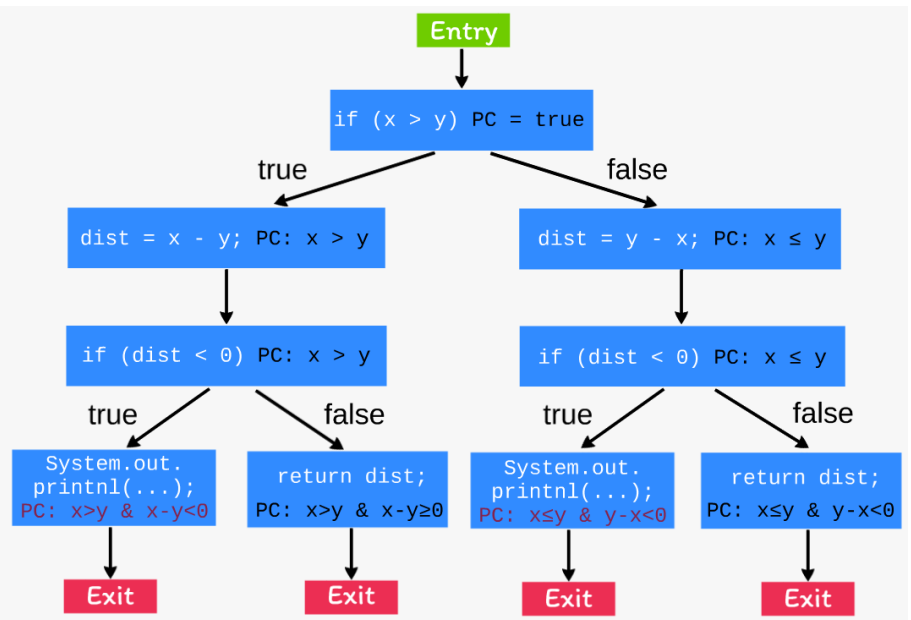
Each state of a symbolically executed program contains a *path condition (PC)*. The path condition is a quantifier-free logical formula over the input symbolic variables with the initial value of true. It accumulates constraints which the inputs must satisfy in order to direct the execution to follow the path associated with the formula.

In addition to maintaining the path condition, symbolic execution engines make use of the so-called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula at any state of the symbolic execution. Practically speaking, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

All of the possible execution paths define a connected and acyclic directed graph called the *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program.

```
1   int distance(int x, int y) {
2       int dist;
3       if (x > y) {
4           dist = x - y;
5       } else {
6           dist = y - x;
7       }
8       if (dist < 0) {
9           System.out.println("Error");
10      }
11      return dist;
12  }
```

(a)



(b)

Figure 1: (a) Sample code that determines the distance between two integers on the number line. (b) The symbolic execution tree of the sample code that handles variable x and y symbolically.

Figure 1 (a) shows a simple Java method that determines the distance between the two integer parameters x and y. The symbolic execution of this code is illustrated in Figure 1 (b) with the corresponding symbolic execution tree, showing the actual path condition. Suppose that parameters x and y are handled symbolically, the initial value of the path condition is true. Encountering the first if statement in line 3, there are two possibilities, namely the logical expression may be true or false; hence the execution branches and the logical expression and its negation are added to the PC.

The value of variable *dist* will be a symbolic expression: $x - y$ on the true branch and $y - x$ on the false one. As a result of the second if statement (line 8), the execution branches and the appropriate PCs are again appended. On the true branches it is obvious that the formulas are unsatisfiable (shown in red), meaning we cannot specify such x and y that meet the conditions. So long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned; then there is no sense in continuing the controversial execution.

2.2 RTEHunter

Now we will give a brief description of our symbolic execution engine called RTE-Hunter in order to explain the optimization approaches and investigations presented in this study.

RTEHunter was created with the goal of detecting runtime errors in Java applications without running the program in a real-life environment. In contrast with other symbolic execution tools [6, 21, 36, 37, 38], generating test cases which lead to failure is not a goal here, but rather to produce a descriptive designation of the execution path that led to a fault.

RTEHunter was developed in C++ and so far the detection of four kinds of runtime faults have been implemented: (1) null pointer dereferences, (2) array over-indexing, (3) array creation with a negative size, and (4) division-by-zero errors.

Instead of starting the symbolic execution from the *main()* method that is the entry point of a Java program, RTEHunter performs the analysis by symbolically executing each method of the system one after the other [20]. This does not mean that the engine cannot handle method calls, as the analysis is interprocedural. The engine handles method calls by placing the actual parameters onto the stack and giving the control to the callee.

The parameters of the method and the referred but not initialized variables are handled as symbols at the beginning of the symbolic execution. It is essential that we only report an error if it is proved that during the execution the value that causes the problem can be determined by constant propagation. In other words, if a method call passes a concrete null value, and RTEHunter finds a path in the called method that dereferences this parameter, we will fire an error, but if the dereferenced variable is a symbol we will not, because its value is unknown or uncertain.

The symbolic execution is performed using the language-dependent *abstract semantic graph (ASG)* [11] of the program by interpreting the nodes of this graph

in a defined order. The order is defined by the language-independent *control flow graph (CFG)* [2], which is constructed for each method. The nodes of the control flow graph are called *basic blocks*. A basic block represents a straight-line piece of code that is guaranteed to execute sequentially (i.e. it does not include any jumps or jump targets) by lining up the appropriate ASG nodes according to the sequential execution. Directed edges between any two basic blocks are used to represent jumps in the control flow. In RTEHunter, for each method analyzed the symbolic execution tree is constructed by traversing the CFG and for each basic block a symbolic state will be created in the tree. Loops and recursions are not handled in any special way and the traversal of the CFG results in simple unrolling. Figure 2 shows the control flow graph constructed for the method *distance()* shown in Figure 1 (a) and the symbolic execution tree that RTEHunter creates using the control flow graph is shown in Figure 3.

Listing 1 shows the pseudo-code of the algorithm used in RTEHunter which builds up the symbolic execution tree while executing symbolically each path with a C++-like syntax. The search-and-build strategy shown here is depth-first search. The construction of the execution tree commences with the method *search()*. Here, we first get the root state of the tree, and initialize the *strategy* object with this. Afterwards, the while loop always gets the next state to be executed. The execution of a state is performed by method called *executeState()*, which interprets the nodes that are in the basic block which the state is created from according to the semantic of the Java programing language. The strategy object provides the next state according to the implemented strategy. In this listing, the strategy is implemented in the class *DepthFirstSearchStrategy*, in which the *getNextState()* method is the essential part of the traversal. It gets the top-most element from the stack and expands this state, intending to get all of its descendants, then it puts them onto the stack. The *expandState()* method of our expander object constructs the child states according to the descendent Basic Blocks in the CFG and the information that is got from the execution of the parent states. For instance, if the parent state represents an if statement that would have two children, but the logical expression could be evaluated by execution of the parent, the *expandState()* will not provide both children, but just the appropriate one. The stack data structure (LIFO queue) provides the depth-first search traversal.

Listing 1: The main algorithm of tree building and execution in RTEHunter.

```
1  Strategy* strategy = new DepthFirstSearchStrategy(expander);
2
3  void search(StateFactory stateFactory) {
4    State *rootState = stateFactory.getRootState();
5    strategy->initialize(*rootState);
6    State* nextState = NULL;
7    while (nextState = strategy->getNextState())
8      executeState(*nextState);
9  }
10
11 class DepthFirstSearchStrategy: public SearchStrategy {
12   private:
13     std::stack<State*> stack;
14
15   public:
16     DepthFirstSearchStrategy(StateExpanderInterface& expander)
17       : SearchStrategy(expander), stack() {}
18
19     void initialize(State& rootState) {
20       stack.push(&state);
```

```
21        }
22
23        State* getNextState() {
24          State* front = stack.top();
25          stack.pop();
26          std::vector<State*> children=expander.expandState(*front);
27          for (State* child : children)
28            stack.push(child);
29          if (stack.empty())
30            return NULL;
31          return stack.top()
32        }
33    };
```

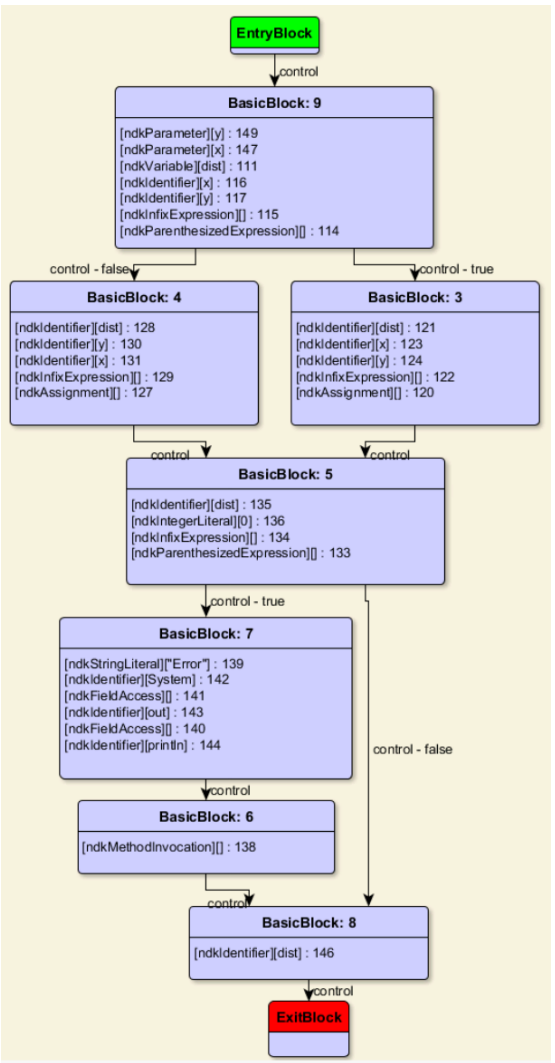


Figure 2: The control flow graph (CFG) constructed for the method in Figure 1 (a).

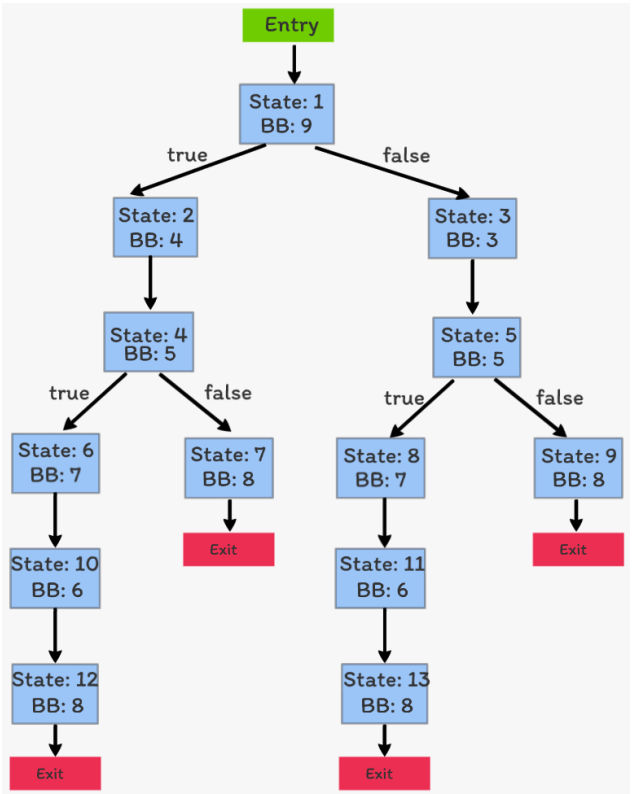


Figure 3: The symbolic execution tree constructed by RTEHunter for the code shown in Figure 1 (a).

RTEHunter is able to handle integer, floating point, reference (including aliasing), and array type data. It models the memory by storing the variables in a special data structure optimized for memory usage. We do not store the whole variable set for each state, but only the changes. That is, if the engine wants to read a variable that was assigned in a parent state, it will be found only in the storage of the appropriate parent, but if the value a variable was changed by the current state, the updated value will be stored in the variable storage of the current state. Moreover, each state has a stack that is used for passing parameters during method invocation and for calculating expressions.

The same runtime failure can be detected repeatedly if multiple execution paths have been explored which led to the same program location. However, it is not obvious how to determine in which cases we consider two errors the same, because the original cause of two errors may be different despite the fact that the detection points are the same. For example, a reference type variable can be set to null at different locations in the program, and then dereferenced at the same place; then

this will cause a *NullPointerException*. The detection location is the same, but the causes (and the possible fixes) may differ. However, by examining the results, we realized that if multiple paths led to the same location, in most of the cases the cause is the same too because the one path is a suffix for the others. Then it seems that if multiple paths lead to the same error location, we should retain only the shorter one and filter out the others.

To limit the size of the symbolic execution trees built up for each method and the maximum depth and the maximum number of states can be specified. In accordance with the above, depth here means basic block depth. Furthermore, it is also possible to define custom search search strategies with which the traversing of the execution tree can be guided.

The output of the RTEHunter contains the errors detected that indicate their type and the execution path by a list of states from the entry point to the exact location where the error occurred. This data are written to a formatted text file, but RTEHunter has also been integrated into the SonarQube quality management platform [8].

3 Approach

3.1 Optimal Maximum Depth and State Number

As we mentioned in Section 2.2, two kinds of constraints can be set up in RTE-Hunter in order to limit the size of the symbolic execution tree and prevent a state explosion. Namely, (1) the maximum depth (which means state depth), and (2) the maximum number of states can both be adjusted. With a given maximum depth, the maximum state number defines the width of the tree. It should be added that these constraints are applied to each symbolic execution tree built for each method of the given system separately and they are not global limits for the system-wide analysis.

In our experiments, we ran RTEHunter with different depth and state number limits on three open source Java systems. The systems analyzed with their total lines of code metric are listed in Table 1.

Table 1: The chosen Java systems on which the measurements were carried out

System	TLOC
ArgoUML	372K
Jetspeed	275K
JFreeChart	329K

In addition to the maximum depth and the maximum state number, the final shape of the symbolic execution tree is also determined by the search strategy applied.

In our experiments we sought to ascertain the optimal depth and state number; and in order to find more runtime errors in less time, we used the default depth-first search strategy.

3.2 Custom Search Strategies

Since the search strategy that is used to direct the traversal influences the shape of the final symbolic execution tree, it plays a significant role in finding those states where runtime errors actually occur.

In the actual stage of the traversal of the state space (i.e. the symbolic execution tree), a search strategy tells us from which state among the current leaf states the exploration has to continue, i.e. which state has to be expanded as the next step of the traversal.

In contrast to depth-first search where the actual leaf states placed into a LIFO (last in, first out) queue, in our custom search strategies a score is assigned to every current leaf, which will be placed into a priority queue. When the engine have to chose a state as the next step, it chooses the one with the highest score to continue the traversal. The implementations of these strategies are very similar to the code of class *DepthFirstStrategy* shown on Listing 1, but the stack member is replaced by a priority queue that orders the states by score, and the top is the one has the highest score.

Next, we will describe two new search strategies which implement heuristics to direct the search towards the potential runtime issues.

3.2.1 The Null-heuristic Search Strategy

This search strategy attempts to drive the traversal to find more null pointer dereference issues. Our motivation of focusing on null pointer dereferences is that according to static analysis the most common checks against exceptions are null-checks in Java sources, implying that this is the most common runtime issue that may occur [12, 32]. We also discovered that this type of runtime error is the most common one that RTEHunter encounters.

For each state we summarize the number of reachable reference-type values (variable values, literals, function return values, etc.), whose value is *null* at the current symbolic state of the given Java program. To continue the traversal, the engine chooses the state with the highest number of null values assigning higher probability value to find possible null pointer dereferences in that state and in the sub-tree obtained from it.

3.2.2 A Linear Regression-Based Search Strategy

We developed a search strategy that supports the detection of not just null pointer dereferences, but also all the four types of issues that RTEHunter is able to detect. To implement such a search strategy we used a linear regression model that assigns a score to each leaf state during the search. The score is the estimated number of

runtime issues that might have been detected in the sub-tree reachable from the state. We chose linear regression because it can be applied on continuous class labels and it provides a relatively quick result for an unseen example.

The training data of the linear regression model contains one training example for each state got from symbolic execution trees that were traversed previously by the engine. The label (i.e. the supervisory signal) for each example is the number of runtime issues that were detected in the sub-tree under the state that the example belongs to.

We defined five attributes as predictors that can be determined for each state. Attributes may containe both static source-code information and dynamic information that the symbolic execution supplies.

The attributes are the following:

1. *The depth of the state in the symbolic execution tree.* If there is a tendency of how deep the significant part of the faults occur, the information will be encoded into the model.

2. *The number of null values in the state*, as described in Section 3.2.1.

3. *The sum of the number of zero numeric type values (variable values, literals, function return values, etc.) in the state, and the number of division operators reachable from the state according to the control flow graph in 15 basic block depth.* Here, we combined the dynamic information of zero values and the static information of the number of division operators in the possible future of the execution. This attribute is a heuristic for finding division-by-zero errors.

4. *The Logical Lines of Code (LLOC) metric of the method that the state belongs to.*

5. *The cyclomatic complexity metric [26] of the method that the state belongs to.*

As lines of code (LOC) and cyclomatic complexity have proved to be promising defect predictors [27, 28], attributes 4 and 5 should be useful in our heuristic. Both of them were calculated using static a source code analyzer tool called *SourceMeter*[1].

We applied the linear regression algorithm implemented in the *Shark* machine learning library [18].

4 Results

4.1 Optimal Maximum Depth and State Number

In order to ascertain the optimal limitations of the symbolic execution tree built by RTEHunter with the goal of finding runtime issues in a minimal time frame, we performed numerous analyses and applied different constraints.

[1] http://www.sourcemeter.com/

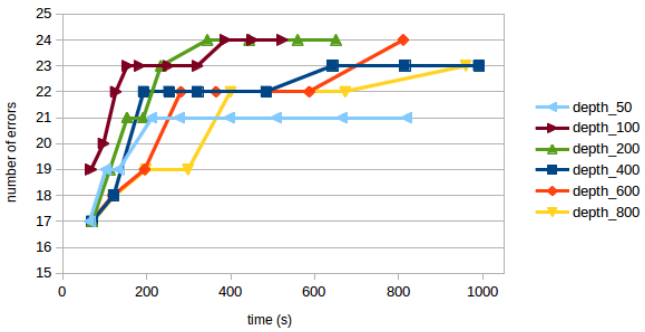


Figure 4: The increase in the number of errors at analysis time using different depth limits in ArgoUML.

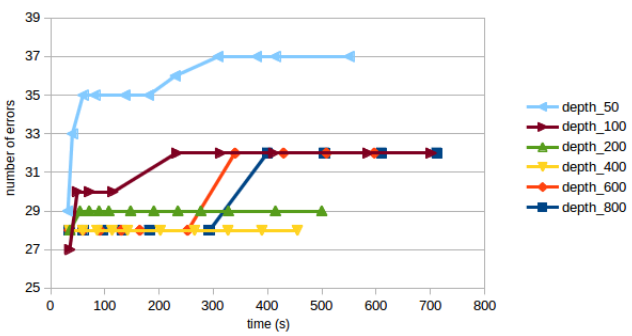


Figure 5: The increase in the number of errors at analysis time using different depth limits in Jetspeed.

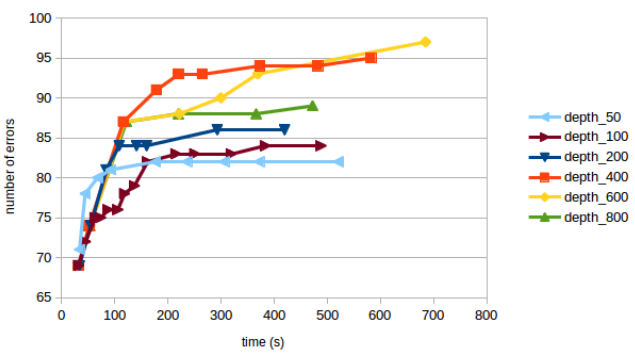
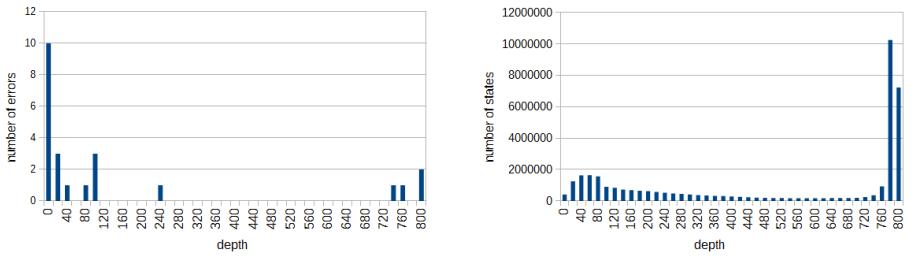
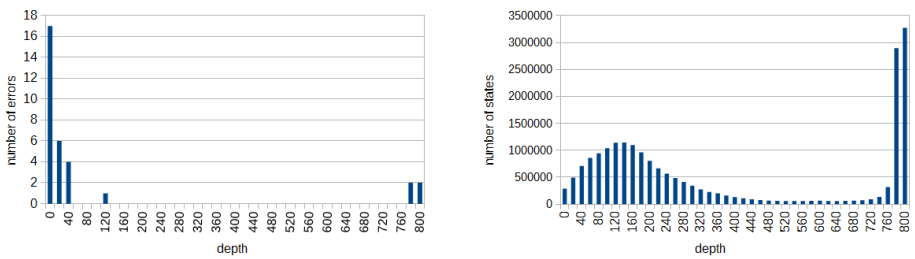


Figure 6: The increase in the number of errors at analysis time using different depth limits in JFreeChart.



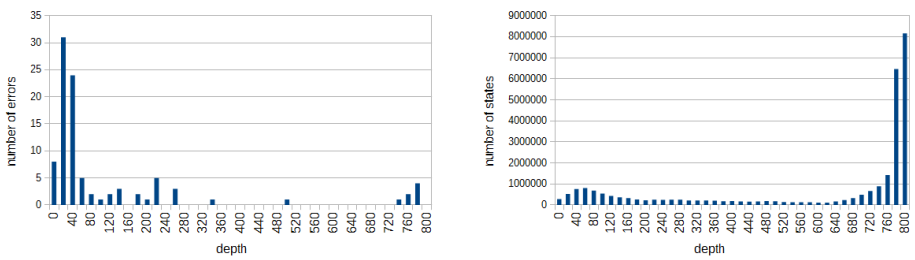
(a) (b)

Figure 7: (a) The distribution of the errors at different depth levels in ArgoUML. (b) The distribution of the states at different depth levels in ArgoUML.



(a) (b)

Figure 8: (a) The distribution of the errors at different depth levels in Jetspeed. (b) The distribution of the states at different depth levels in Jetspeed.



(a) (b)

Figure 9: (a) The distribution of the errors at different depth levels in JFreeChart. (b) The distribution of the states at different depth levels in JFreeChart.

First of all, we found that the number of executed states is closely correlated with the analysis time. The Pearson-correlation coefficients are all above 0.99, which is a strong positive correlation, and it means that the high state limit corresponds to a high run-time. The results are significant at $p < 0.05$. However, the correlation coefficients among the number of states and the number of issues found range from 0.3 to 0.8, indicating a weaker relationship, and the results are not significant in many cases at $p < 0.05$. Hence the number of states seems to determine the execution time, but the number of errors probably depends on other factors as well.

To understand the role of maximum depth, we ran RTEHunter with different depth limits, at each depth limit level and with different maximum state sizes, and this allowed us to investigate how the results varied by increasing the analysis time. The depth limits chosen were 50, 100, 200, 400, 600 and 800, for each the maximum state values differ from 200 to 10,000. The results were put onto line diagrams shown in figures 4, 5 and 6 for ArgoUML, Jetspeed and JFreeChart, respectively. The diagrams show how the number of errors grows in time at each depth limit level. On each line the dots represents an RTEHunter run with a specific state number limit, which are formed to extend the analysis at least nearly 500 seconds.

In general, each line starts increasing, and after a while the number of errors does not increase anymore. The reason why the number of errors stagnate after a time may be because all of the errors were found at that depth level, and to detect new ones the analysis needs to go deeper. The depth limit is considered to be better which rises rapidly and with which the engine detects higher number of errors. The best depth limit varies from system to system.

As regards ArgoUML, the 100-depth configuration increases the most rapidly, but at the depth of 200 one can find 24 errors in slightly shorter time. By decreasing the depth limit to 50, the results get worse, and the 400- 600- and 800-depth configurations also perform worse.

With Jetspeed, the 50-depth limit is far better than the others. While with the depths of 100, 600 and 800 we can reach only 32 errors under 500 seconds, we managed to detect 37 errors with the 50-depth configuration. These results suggest that the majority of the errors in Jetspeed are at a depth of below 50.

Among the investigated depth limits, the 400-depth may be considered to be the optimum in the case of JFreeChart. However, the 50-depth one starts to rise the most rapidly, then it stops growing after 100 seconds. After 600 seconds the 600-depth limit becomes slightly better than the 400-depth one, but in general the 400-depth one performs the best.

To understand why the above-mentioned depth-limits performed the best in the experiments, we analyzed the distribution of the errors at different depth levels. Subfigures (a) of figures 7, 8 and 9 show that the number of errors found at each depth level. The depth-level intervals formed to be 20-length intervals in each diagram. The height of the bars in each diagram represents the number of errors found in 20-length depth intervals. The data is derived from an 800 depth and 15,000 state limit run, with which we attempted to analyze as big symbolic

execution trees as the memory consumption made possible.

In general, the same pattern appears to be present in all three systems. The number of errors are significant at shallower levels, then in the middle just a few of them were detected, and close to the depth limit errors occur again but not at such high numbers as in the shallower levels. E.g. the error distribution of Jetspeed in Figure 8 (a) tells us that the majority of the errors were found at a depth limit of 40 or less, which explains why the 50-depth limit is so satisfactory in Figure 5. Although deeper configurations reached more states, the error density is rather low there, hence we wasted time spent in executing these.

We have also plotted the distributions of all the states that were explored by RTEHunter in figures 7, 8 and 9 (b). These diagrams show the overall shape of the symbolic execution trees traversed through the analysis. A similar pattern can be seen in the error distribution diagrams, which partially explains the error distribution: at depth levels where more states are explored, more errors can be found. However, there are many more states near the 800-depth maximum-limit than in the shallow parts of the tree, but the number of errors is higher in the shallow levels than at the bottom. This leads us conclude that the the runtime issues that RTEHunter can detect are in general more common between levels 0 to 60 basic block depths compared to the deeper levels.

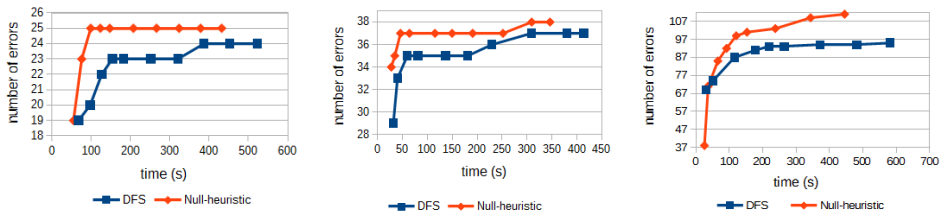
It should be added that the search strategy which is used to explore the state space has a marked effect on the state distribution, and hence on the error distribution too.

4.2 Null-heuristic Search Strategy

The goal of the null-heuristic search strategy is to increase the number of runtime issues detected in the given time frame, compared to the default depth-first search. In particular, we focused on the number of null pointer dereferences. To make a comparison, we used those configurations which were found to be the best in Section 4.1 as a reference. The maximum depth for ArgoUML is 100, 50 for Jetseed and 400 for JFreeChart. We also chose the same state number limit sequence to expand the analysis time as before. With these parameters, but this time with our novel null-heuristic we repeated the experiments. The results of these experiments are shown in Figure 10.

In each case the null-heuristic approach performed better, because the number of detected issues increases more rapidly and the values higher, i.e. it found more runtime issues in less time which surely confirms the efficiency of our algorithm. It is worth mentioning here that over 90% of the errors found in these systems are null pointer dereferences, and the new search strategy applied does not modify this ratio significantly.

What is more interesting is that the same analysis sequence with the null-heuristic approach finished faster than before. E.g. in the case of ArgoUML the last analysis (the last dot on the line) lasted 431 seconds with the null-heuristic, and 522 seconds with the conventional DFS. This point is surprising because we need to calculate the number of null reference values for each state and also maintain



(a) (b) (c)

Figure 10: The efficiency of null-heuristic search compared to the default depth-first search on ArgoUML (a), Jetspeed (b), JFreeChart (c).

a priority queue to keep the leaves in for the null-heuristic algorithm. Probably the reason why it is still faster is that it guides the search towards states whose execution time is shorter.

4.3 Linear Regression Based Search Strategy

Table 2: The number of detected runtime issues with the linear regression based search strategy (LR based) compared to the default depth-first search (DFS)

		500 max state	1000 max state	1500 max state
ArgoUML	DFS	22	23	23
(max depth: 100)	LR based	51	54	55
Jetspeed	DFS	33	35	35
(max depth: 50)	LR based	66	74	73
JFreeChart	DFS	91	93	94
(max depth: 400)	LR based	122	131	138

In the evaluation of the linear regression-based search strategy, we use 10-fold cross-validation on each system in the following way. Firstly, to form the folds we sort the methods of the system by lines of code (LOC). In the sorted list, each $j^{th} method$ is placed into $fold_i$ if equation $j \ mod \ 10 \ = \ i$ is satisfied. In other words, every tenth method will go to the same fold (see Figure 11), ensuring that no fold differs too much from the others in the length of the methods contained.

After running RTEHunter on 10 folds, we summarized the number of errors that were detected in each fold. The structure of the folds ensures that each error is counted only once. The number of errors found using this strategy (LR-based) is shown in Table 2 and it is compared to the errors found by the default depth-first search strategy (DFS). With all the subject systems we examined the depth limit that was found to be the best in Section 4.1 using DFS: 100 for ArgoUML, 400 for JFreeChart, and 50 for Jetspeed. As regards the maximum state number constraint, we provide results for 500, 1000 and 1500 maximum state values.

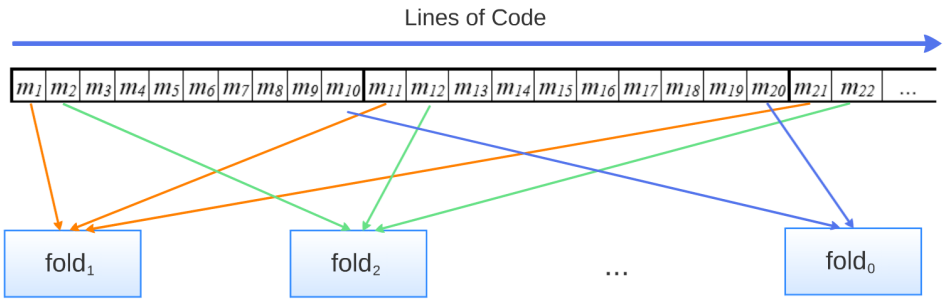


Figure 11: The formation of the folds used to perform 10-fold cross validation.

As the results demonstrate, our novel algorithm outperforms the default one in each case. In ArgoUML and Jetspeed, we found more than twice as many errors as we did with DFS. However, in the case of JFreeChart this ratio is smaller, the difference being still significant: it formed around 30 to 40 more issues were discovered.

The reason why we do not present the runtime here is due to the implementation details of the 10-fold cross validation. Currently, before RTEHunter starts the analysis of each fold, it has to load the ASG and then rebuild the CFG for architectural reasons. This introduces an overhead, which is not present in the case of a conventional run. Apart from this shortcoming, the difference in the number of detected issues is still significant.

5 Threats to Validity

In the present study we do not focus on the precision of RTEHunter. Some of the issues found may be false positives and these may change or invalidate the result of our investigations. Manual validation needs a considerable amount of work in the case of such a high number of errors presented here. However, in a future work we plan to perform the manual validation and repeat the experiments with the updated dataset. We also intend to examine how the false positive rate is affected by the traversal strategy because we wish to avoid situations where a deeper analysis on a path produces more false positives for instance because of an incorrectly handled coding pattern.

6 Related Work

Symbolic execution engines and similar tools. The idea of symbolic execution is not new, and the first publications and execution engines appeared in the 1970's. One of the earliest papers is by King, which laid down the fundamentals of symbolic execution [21] and presented the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers

symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another paper from the 1970's is by Boyer et al., who presented a similar system called SELECT [3] that can be used for executing LISP programs symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation and in addition, for each path it returns the path condition over the symbolic variables.

Starting from the last decade interest in this technique has been steadily growing, and numerous programs have been developed that seek to dynamically test input generation using symbolic execution. There are also approaches and tools for generating test suites for .NET programs using symbolic execution. Pex [36] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [19], which seeks to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

In Section 2.2, we mentioned that the results of RTEHunter can be published into a SonarQube instance [8]. SonarQube has its own symbolic execution engine for Java language, which similar to us is intended to find tricky bugs that are almost undetectable by developers unaided. It is able to find three kinds of issues compared to our five: (1) null pointer dereference, (2) unclosed resource and (3) a rule named "condition should not unconditionally evaluate to true or false". Based on our thorough testing of the tool it has a constraints solver, but it does not handle function calls, meaning it might miss serious errors like null pointer dereferences that RTEHunter is able to detect even if the control flows through multiple function calls. Another difference with RTEHunter is that if an issue is found, SonarQube presents only that part in the source code where the issue actually appears, and it does not give the path that leads to the fault which can really help the developer in the debugging work. It should be added that widely used static analysis tools like PMD [1] and FindBugs [17] can also be integrated into SonarQube to find issues and bugs. PMD performs syntactic checks on the source-code by building an ASG first and it does not carry out any deeper analysis to detect runtime issues like a symbolic execution tool. Beyond syntactic checks, FindBugs syntactically matches source code to known suspicious programming practice and also uses data-flow analysis to check for bugs like null pointer dereference, class cast exception and array index out of bounds exception in an intraprocedural way, but this approach does not present the results as an interprocedural symbolic execution.

The handling of the state explosion in symbolic execution. To reduce the state space of the symbolic execution, Symbolic PathFinder [30] based on the Java PathFinder model checker offers a number of options. Similar to RTEHunter, the

maximum depth of the symbolic execution tree can be specified, and the number of elementary formulas in the path condition can be restricted. Another possibility is that with options we can restrict the value ranges of the integer and floating point type symbolic variables. In addition, Symbolic PathFinder lazily initializes object references and uses types to infer aliasing.

Bucur et al. [4] addresses the problem of path explosion by parallelizing symbolic execution in such a way that it scales well on large clusters of cheap commodity hardware. The system, called Cloud9, can automatically test real systems that interact in complex ways with their environment.

Another approach for reducing the state space, presented by Chipounov [9], is not to execute the whole program symbolically, but just portions of it. The engine can start the symbolic execution at arbitrary places of a whole system, including applications, libraries, the operating system, and device drivers. It seamlessly goes back and forth between symbolic and concrete execution, while transparently converting system states from symbolic to concrete and back. A similar approach is used for testing NASA Software [31]. The idea is also to start the program in normal mode as in a real-life environment, then at given points (e.g. at more complex or problematic parts in the program) it can switch to the symbolic execution mode. The CUTE and jCUTE systems [33] constructed by Sen and Agha, are also concolic executors, where they start at an arbitrary function by initializing pointers based at first on a simple heap with abstract addresses and incrementally increase the heap complexity in subsequent runs.

In order to make symbolic execution more scalable, Majumdar and Xu propose using symbolic grammars to guide symbolic execution by reducing the space of possible inputs [25]. Godefroid et al. utilized a similar approach. They set up the grammar-based specification of highly-structured inputs of symbolic execution such as compilers and interpreters [14].

Another approach is to reuse and merge the paths that were explored earlier. Also, reusing the analysis of lower-level functions in subsequent computations improves the scalability of symbolic execution [13].

Loops and recursions with conditions that cannot be evaluated during the symbolic execution result in infinite constructs that can explode the state space without benefit. For this reason, the handling of these constructs may have significant effects. CBMC is a Bounded Model Checker for C and C++ programs [10]. CBMC is able to verify array bounds, exception handling correctness, pointer safety and user defined assertions as well. In bounded model checking, the potentially infinite constructs (e.g. while loops, recursion) are unwound only n times, where this number n is the upper bound. CBMC pre-processes the program into an equivalent program that uses only while, if, goto statements, and assignments. Next, all while loops are unwound using the following transformation n times:

while (cond) instruction; $\rightarrow$ *if (e) {instruction; while(cond) instruction}*

The last while loop is replaced by assertion *!cond*, which ensures that the program never performs more iterations. This unwinding assertion is verified along with the user defined assertions, and if it fails, one more iteration is required in the unwinding. Afterwards, the program only consists of if instructions, assignments,

assertions, labels, and forward goto instructions, and it is then transformed into static single assignment (SSA) form, from which a bit vector equation is created together with the target rule to be verified. If this equation is satisfiable, the tool finds a violation. The mechanism presented here of unrolling loops only a certain number of times would be a good idea to integrate it into RTEHunter to reduce the state space generated by loops. Currently, we use a simple unrolling until we reach the depth or the overall state limit. Another strategy would be to recognize patterns in the basic block sequence during a loop unwinding. For example, when a basic block is visited too many times, the execution is deep inside a recursion or a loop. This strategy works well for the Clang Static Analyzer [22].

Search heuristics in symbolic execution. One of the key mechanisms used by symbolic execution tools to prioritize path exploration is the use of search heuristics. Most heuristics focus on achieving high statement or branch coverage, but they could also be employed to optimize other desired criteria. The main difference compared to our study is that we optimize for execution time and also for the number of detected issues.

KLEE [6] is another symbolic execution engine that seeks to automatically generate tests that achieve high coverage. Similar to our approach, it is possible to use various heuristics to prioritize the most interesting paths first. KLEE selects the next state to run by interleaving the two search heuristics, namely random state selection and a coverage-optimized search that attempts to select states likely to cover new code. Currently, our search heuristics do not incorporate any test coverage information, but in the future we intend to examine this possibility. Random exploration proved to be an efficient test generation approach by Burnim et al [5] as well.

The EXE (EXecution generated Executions) [7] presented by Cadar et al. at Stanford University is an error checking tool designed for generating input data on which the program terminates with failure. The heuristic in EXE favors previously visited statements that were run the least number of times.

The AUSTIN tool applies fitness functions to drive an evolutionary search of the test input space with dynamic symbolic execution [23].

Ma et al. [24] focus on debugging scenarios when the developer already knows about the faulty line, but they might not know exactly how to reproduce the failure or even whether it is reproducible. The approach also applies search strategies that try to direct the symbolic execution to the target line. One strategy is the *shortest-distance symbolic execution (SDSE)*, where we pick the path that currently has the shortest distance to the target line according to the control flow graph (CFG) of the program. The other one starts at the target line and works backward until it finds a realizable path from the start of the program, using standard forward symbolic execution as a subroutine.

In general, our problem lies in the domain of Search-Based Software Engineering (SBSE), where search-based optimization algorithms are used to address problems in software engineering - e.g. to figure out the smallest set of test cases that cover all branches in this program or the set of requirements that balances software

development cost and customer satisfaction. Harman et al. give a comprehensive survey addressing this area [16]. In our case, we look for the symbolic execution tree that has smallest exploration time and covers the greatest number of runtime issues.

7 Conclusions and Future Work

The main goal of this study was to optimize the RTEHunter symbolic execution engine to detect more runtime issues in less time. Because of the path explosion problem, the limitation of the state space of the symbolic execution assumes a major importance in this scenario. In the empirical investigations on three open-source Java systems, it turned out that adjusting the maximum number of states for the symbolic execution trees has an effect on the execution time, but not on the number of issues found. However, the constraints on the depth of the tree is more important in the detection of runtime errors. We found different optimal depth limits for the three different systems, but we can say that errors occur more often at the basic block depth of 0 to 60 compared to the deeper levels in the systems we analyzed, but it also strongly depends on the search strategy that is applied.

Here, we propose two novel search strategies that seek to guide the symbolic execution towards the more error prone source-code fragments using both static and dynamic information. The null-heuristuc search strategy performs better by finding up to 16 % more errors within the same time frame than those found using the default depth-first search. The linear regression-based heuristic also outperforms DFS, and it can detect over twice as many errors in ArgoUML and Jetspeed.

In the future, we would like to include more systems into the empirical experiments and this may help us to find the optimal depth and state number limits of symbolic execution tree in general, which should make RTEHunter and other symbolic execution engines more efficient. We also plan to develop more efficient search strategies, by adding new features and applying other machine learning approaches. Also, we think that the investigation of including test coverage information in the search heuristic might be a promising approach. We would also like to manually validate the errors reported by the RTEHunter to discover its precision in practice.

In this study we did not place any emphasis on the practical usage of the search strategies developed in a real-life product or examine which strategy is good for which type of error. The null-heuristic search might be good for finding null dereferences, but it might also degrade the quality of detecting other types of issue. In real-life applications, one option might be to develop specific, well performing search strategies for each issue type. However, it may be time consuming to rerun the symbolic execution for each type. Another option might be to develop one complex search strategy with general predictors that performs well for all or for most of the issues. This approach might require less time because we need to construct the state space only once, but it might not scale well as the number of checks increase or do not perform as well as the issue-specific heuristics. We plan to investigate this in more detail as well.

8 Acknowledgment

My sincere thanks goes to Dr. Rudolf Ferenc. Without his precious support and help, it would not have been possible to conduct this study.

References

[1] Pmd/java. `https://pmd.github.io/`. Accessed: 2017-03-21.

[2] Allen, Frances E. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] Boyer, Robert S., Elspas, Bernard, and Levitt, Karl N. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[4] Bucur, Stefan, Ureche, Vlad, Zamfir, Cristian, and Candea, George. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[5] Burnim, J. and Sen, K. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Cadar, Cristian, Dunbar, Daniel, Engler, Dawson R, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., and Engler, Dawson R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[8] Campbell, G. Ann and Papapetrou, Patroklos P. *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.

[9] Chipounov, Vitaly, Georgescu, Vlad, Zamfir, Cristian, and Candea, George. Selective Symbolic Execution. In *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[10] Clarke, Edmund, Kroening, Daniel, and Yorav, Karen. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, pages 368–371. ACM, 2003.

[11] Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, IEEE Computer Society, oct 2002.

[12] Flanagan, Cormac and Leino, K. Rustan M. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[13] Godefroid, Patrice. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[14] Godefroid, Patrice, Kiezun, Adam, and Levin, Michael Y. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[15] Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[16] Harman, Mark, Mansouri, S Afshin, and Zhang, Yuanyuan. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[17] Hovemeyer, David and Pugh, William. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[18] Igel, Christian, Heidrich-Meisner, Verena, and Glasmachers, Tobias. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[19] Jamrozik, Konrad, Fraser, Gordon, Tillman, Nikolai, and Halleux, Jonathan. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.

[20] Kádár, István, Hegedűs, Péter, and Ferenc, Rudolf. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[21] King, James C. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[22] Kremenek, Ted. Finding software bugs with the clang static analyzer. *Apple Inc*, 2008.

[23] Lakhotia, Kiran, McMinn, Phil, and Harman, Mark. An empirical investigation into branch coverage for c programs using {CUTE} and {AUSTIN}. *Journal of Systems and Software*, 83(12):2379 – 2391, 2010.

[24] Ma, Kin-Keung, Phang, Khoo Yit, Foster, Jeffrey S., and Hicks, Michael. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Majumdar, Rupak and Xu, Ru-Gang. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA, 2007. ACM.

[26] McCabe, Thomas J. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[27] Menzies, Tim, Greenwald, Jeremy, and Frank, Art. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[28] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[29] Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Science/Engineering/Math, November 2001.

[30] Păsăreanu, Corina S. and Rungta, Neha. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[31] Păsăreanu, Corina S., Mehlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen, Lowry, Michael, Person, Suzette, and Pape, Mark. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[32] Ryder, Barbara G., Smith, Donald, Kremer, Ulrich, Gordon, Michael, and Shah, Nirav. A Static Study of Java Exceptions Using JESP. In *Proceedings of the Ninth International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2000.

[33] Sen, Koushik and Agha, Gul. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.

[34] Song, JaeSeung, Ma, Tiejun, Cadar, Cristian, and Pietzuch, Peter. Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, 2011.

[35] Tassey, G. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.

[36] Tillmann, Nikolai and De Halleux, Jonathan. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Visser, Willem, Păsăreanu, Corina S., and Khurshid, Sarfraz. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[38] Xie, Tao, Marinov, Darko, Schulte, Wolfram, and Notkin, David. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

Acta Cybernetica **23** (2017) 599–613.

Image Processing-based Automatic Pupillometry on Infrared Videos

György Kalmár[a], Alexandra Büki[b], Gabriella Kékesi[b], Gyöngyi Horváth[b], and László G. Nyúl[a]

Abstract

Pupillometry is a non-invasive technique that can be used to objectively characterize pathophysiological changes involving the pupillary reflex. It is essentially the measurement of the pupil diameter over time. Here, specially designed computer algorithms provide fast, reliable and reproducible solutions for the analysis. These methods use a priori information about the shape and color of the pupil. Our study focuses on measuring the diameter and dynamics of the pupils of rats with schizophrenia using videos recorded with a modified digital camera under infrared (IR) illumination. We developed a novel, robust method that measures the size of a pupil even under poor circumstances (noise, blur, reflections and occlusions). We compare our results with measurements obtained using manual annotation.

Keywords: pupillometry, ray propagation, energy attenuation

1 Introduction

In medical diagnostics, non-invasive techniques and their applications are actively studied today. Most of the well-known ones rely on computer aided imaging, such as fMRI, CT, SPECT, and PET. One of the simpler techniques is pupillometry, which can be used to analyze diseases and their pathophysiological changes involving the pupillary reflex. Essentially, it is the measurement of the diameter of a pupil during light stimuli, which induce pupillary light reflex, i.e. the contraction of a pupil in response to light. Pupillary light reflex-derived metrics can be used to monitor the extent of neurological diseases and their response to therapy. Earlier studies examined the parasympathetic function within the context of schizophrenia through the use of pupillometry. These investigations demonstrated that pupil diameter is larger and the reflex is slower in patients with schizophrenia. The

[a]Department of Image Processing and Computer Graphics, University of Szeged, E-mail: {kalmargy, nyul}@inf.u-szeged.hu

[b]Department of Physiology, University of Szeged, E-mail: {buki.alexandra, kekesi.gabriella, horvath.gyorgyi}@med.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.10

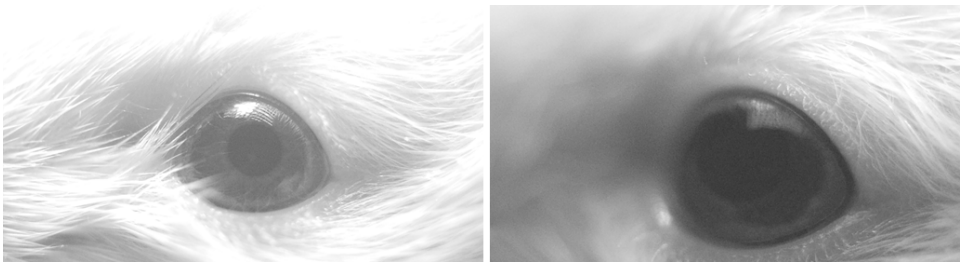


Figure 1: Example for a good (left) and a bad (right) quality image.

basic idea of pupillometry is to record the pupillary reflex with a camera and after the controlled experiments, the video recordings are processed. Measuring the diameters manually frame-by-frame with the aid of a simple program is time consuming and non-reproducible in cases where the number of subjects is high. Automated, specially designed computer algorithms provide faster, more reliable and reproducible solutions for the analysis. These methods use a priori information about the darkness and circular shape of the pupil. From an image processing perspective, the problem is essentially that of circle detection. A combination of well-known circle detection methods with special, pupillometry-related information should result in accurate and robust solutions.

Our study focuses on measuring the diameter of rats' pupils using infrared (IR) videos recorded by a digital camera with an IR filter. Medical research supported by our efforts investigates the influence of schizophrenia on pupillary light reflex. During the experiments, rats are held by hand on a desk while a short light impulse is shone into their eyes. The camera with a fixed focal length is properly positioned prior to the experiments to be able to record the response of pupil to the light stimulus. The videos, however, have certain quality defects. Breathing and slight movements of the rats cause scattering movements and significant blur, which makes the segmentation of pupil impossible with simple binarization. Beyond these artifacts, the rats' eyes have a red color, without any melanocyte, resulting in a low contrast between the pupil and the iris. Low lighting conditions force a higher ISO level value during the recordings, and this leads to noisier images. The reflections of the illuminating IR LED source obscure a large part of the pupil boundary, while other overlying entities such as the whiskers of the animal can make the pupil virtually undetectable. In Figure 1 two sample images for a good and a bad quality video are shown. On the left hand side the dark region in the middle of the eye is the pupil region, which is easy-to-detect, in contrast to the picture on the right hand side; the image is blurred, has a low contrast and the pupil region is barely distinguishable from the iris region.

To overcome the above-mentioned difficulties and reliably, accurately detect the pupil, and automatically measure its parameters, we developed a novel ray propagation-based method with energy attenuation. It can find the fine contrast changes at the boundary of the pupil, while tolerating noise, reflections, occlusions

and give a good result even with blurred images. The proposed technique can measure the diameter of a pupil with a small size and sharp edge and also big, blurred, noisy ones using the same procedure.

The paper is organized as follows. In Section 2, related works are mentioned, then in Section 3 we present our proposed automated procedure. In Section 4, we discuss in detail the functionality of the energy-based technique. Then, in Section 5, we elaborate on our estimation and measuring procedure. This is followed by our evaluation and presentation of results in Section 6. Lastly, in Section 7, we summarize our findings and make suggestions for future study.

2 Related works

In image processing, there are many well-known sophisticated circle detection algorithms based on various concepts. One of the most popular approaches is based on the Hough transform. The Circle Hough Transform (CHT) detects circles with a given radius. It uses a 3-D parameter space and local maxima searching in a so-called accumulator space. The input for the CHT is an edge image. For each edge point, it increments the accumulator values that correspond to points located at a given distance from the selected edge point. The accumulator space is a 3-D space. Two dimensions correspond to the coordinates of the center of the circles, and the third to the radius. After processing all edge points, local maxima appear at the accumulator points that correspond to the circle centers. Modifications to the CHT (use of edge orientation, use of a single accumulator space for multiple radii, use of phase to code radii, randomization, etc.) were introduced to reduce the computational cost and improve the detection rate. For more details see [2, 10, 17]. We cannot apply a CHT directly because there may be a substantial blur, part of the pupil boundary may not be observable in the edge image, and the pupil shape may not be a perfect circle.

Another idea is based on the fact that the image gradients at the circle boundary point outwards from the center of the circle (dark circle, white background). In [14], a fast circle detection algorithm is described. First, the gradient image is calculated and because of the symmetry of the circle, for each vector there will be a pair of vectors in opposite directions. For a given vector $\mathbf{V}$, its pair vector $\mathbf{W}$ satisfies two basic conditions; namely the absolute difference between the two directions should be nearly 180 degrees and the angle between the vector $\mathbf{V}$ and the line connecting the bases of vectors $\mathbf{V}$ and $\mathbf{W}$ should be nearly 0 degrees. In the next phase, the corresponding vector pairs are formed. Then, candidate circles are computed for all vector pairs. Afterwards, suitable circles are extracted from the candidates (accumulator matrix, clustering). In our case, the reflections and blurry boundary do not generate a consistent gradient vector space to be able to detect correctly the circular shaped pupil.

In recent years, optimization-based methods have become quite popular. These use well-known optimization techniques to find local extrema in an appropriately chosen function space, some of them having been inspired by biological notions.

Such approaches include the Genetic Algorithms (GA) [1], Bacterial Foraging Algorithm Optimizer (BFAO) [8], Harmony Search Optimization [4], Artificial Bee Colony (ABC) optimization [6], and several others [5, 7].

Pupillometry is a long-known method that is used to objectively characterize the pupil's response to stimuli. Eye tracking, pupil center detecting, and measuring th pupil diameter form parts of an automated pupillometry process. Simpler, earlier methods try to determine a suitable threshold value and binarize the eye image, which is processed by shape analysis or ellipse fitting algorithms [3, 11, 12, 13, 15]. In [9], the authors propose a fully automated procedure for pupil segmentation based on the level set theory. The level set formulation can handle the complex topology of biomedical images regardless of the initial level set configuration. They used a 4-level segmentation, that is suitable for extracting the pupil from the eye picture and measure various morphological parameters, such as the pupil's diameter, centroid and area. The authors of [18] developed an algorithm that utilizes the curvature characteristics of the pupil boundary to determine the visible portion of the pupil, which provides improved estimates of the pupil center when it is obscured by a host of artifacts. The curvature algorithm discriminates between edge points that lie on the smooth pupil boundary and those that lie on the intersection of the pupil with eyelids, eyelashes, corneal reflections or shadows. The non-occluded boundary points are used as input to an ellipse-fitting procedure that offers a robust estimate of the pupil center. The above-mentioned papers describe acceptable solutions for the pupil detection problem, but these methods have their shortcomings when it comes to handling reflections, blur and low contrast difference.

3 Pipeline of the automated pupillometry

Our procedure seeks to support medical staff in their work. In these studies on the connection between schizophrenia and pupillary reflex, they need to analyze sets of videos. Processing hundreds of videos by hand is time consuming and non-reproducible. To overcome this problem, we developed an automated system that can bring a significant speed-up in the processing stage. The input of this system is a video, which contains the recorded pupillary response to a single stimulus. The output is a diagram, that expresses a change in the pupil diameter over time. The pipeline of the proposed automated process is shown in Figure 2.

3.1 Movement filtering

As we mentioned above, the test subjects breathe and make small movements because they have only been slightly sedated and are held by hand. This causes scattering movements of the eye in the videos. To compensate for this artifact, we stabilize the recordings with the help of the Kanade-Lucas-Tomasi (KLT) point tracker [16]. We assume that the size and shape of the eye do not change (rats do not blink during the recordings), and only translational transformations occur between the consecutive frames. The KLT algorithm tracks corner points throughout the

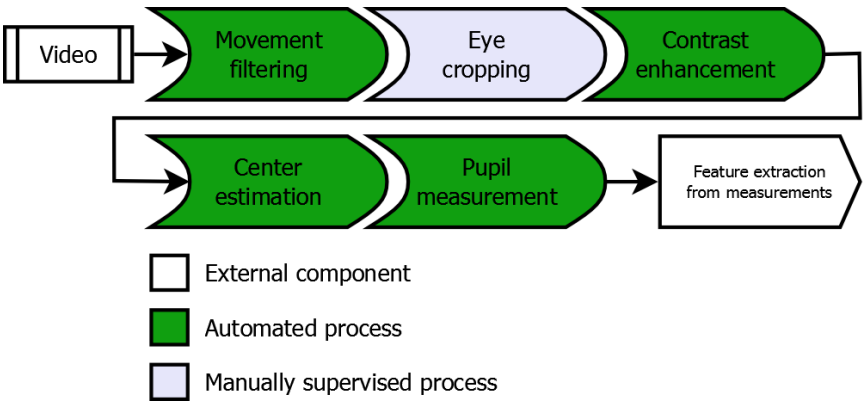


Figure 2: Pipeline of the automated pupillometry.

video. And in each frame the successfully tracked points and their initial positions are connected by translational vectors. The median value of translations is then used to translate the whole frame back to the assumed initial position. The result of this stage is a video, where the eye of the rat has been stabilized.

3.2 Eye cropping and contrast enhancement

The region of interest (ROI) in our case is, of course, the eye region. The diameter of the pupil is expressed relative to the size of the iris. The boundary of the eye is not defined exactly because of shadows and hairs. Experience is necessary to select the appropriate region that just contains the eye. We generate an initial guess based on the projections of dark pixels to the horizontal and vertical axes, but supervision is required. We developed a simple graphical user interface to assist the selection of the ROI. After defining a bounding box that contains the eye, we crop that area from each frame, because we assume that after the stabilization process the eye's position will not change. This in turn reduces the amount of pixel data that will be processed.

One of the major problems is that the contrast between the pupil and iris is very low. The contrast enhancement process attempts to increase the difference by assuming that the minimal pixel intensity within the eye region corresponds to the pupil region. After inspecting the intensity histogram of the eye area, it is clear that it is bimodal. Using this information we can calculate an optimal threshold value that separates the iris region and pupil region. The minimal pixel intensity and the computed threshold value give the two main points for a histogram stretching algorithm.

3.3 Center point estimation

In the next phase, we need to get a good estimation of the pupils's center point. In this phase, the entire video is processed and the output is a position (in two coordinates) that most likely corresponds to the center of the pupil region. This process heavily relies on our novel ray-propagation-with-energy-attenuation technique, which is described below. The center point estimation will be discussed in more details in Section 5.

3.4 Pupil measurement

Lastly, the diameter of the pupil is determined for each frame. We fit an ellipse on the filtered boundary points produced by the novel energy attenuation model-based ray propagation method. The output of the process is a diagram which expresses the change of pupil's parameters during the experiments.

4 Ray propagation with energy attenuation

In the IR videos the contrast between the pupil and iris region is low. One of the main reasons is that the rats' eye contains no melanocyte. Another is that the recordings are made with a modified digital camera with IR filter which requires the use of an intense external IR light source. The relative position between the rat and light source is not constant, and this affects the amount of light reflected from the iris. These factors lead to a significant variation in the video quality and make the proper pupil detection challenging. The proposed method can handle the dynamic change of intensity levels, not only among videos, but also frame-by-frame; and it remains robust to noise.

In the following, we shall assume that the pupil region has a darker color (lower intensity) than the surrounding iris region and that the video frames contain only one dark circular region which is enclosed by a brighter region. It does not matter that only a part of the pupil may be visible or it is affected by significant amount of noise and blurring. We also assume that the center of pupil has been accurately estimated and that it remains the same during the recordings.

The proposed method uses notions and ideas taken from the physics of wave (ray) propagation. The rays have an initial energy which is gradually absorbed by the medium during the propagation because of scattering. The amount of energy loss is proportional to the attenuation coefficient of the material. A medium with no attenuation is called a vacuum. Relating to these principles, the idea is to cast rays from a point and use pixel intensities as measures of attenuation capabilities, and trace them while traveling through the image, then use the information of energy loss characteristics to learn more about the structure of surrounding regions. Higher pixel intensities mean a higher attenuation.

Rays with initial energies radiate uniformly from the estimated pupil center. We trace them until their energy is completely absorbed and record the position of the endpoint ε, where it occurred. Because of the above-mentioned notions are

inherently related to direction and distance from a center point, positions will be determined in the polar coordinate system, whose origin is the estimated center point of the pupil and the polar axis is drawn horizontal and pointing to the right. We will denote the angular and the radial coordinates by θ and r, respectively. At a point $[\theta, r]$, the pixel intensity is $I_{\theta r}$. For a ray with direction θ, the radial coordinate of its endpoint will be denoted by ε_θ.

The basic idea is to treat the pupil region as a vacuum. For each frame, we calculate the vacuum intensity

$$I_v = \frac{1}{|R|} \sum_{x,y \in R} I_{xy}.$$

Here, I_v is defined as the average pixel intensity of a small region R around the estimated pupil center. We shift the original pixel intensities I to I^* as follows: $I^* = \max[I - I_v\ , 0]$. If a ray with energy E travels through a pixel at $[\theta, r]$ with intensity $I^*_{\theta r}$, we can update the current ray energy using

$$E := E - f(I^*_{\theta r}).$$

We will use a quadratic function for f to emphasize the contrast between the pupil and iris.

Let us define the set of initial energies $E_I = \{e_k = \alpha + k\delta \mid k \in \mathbb{N}\}$, where α is an initial offset energy and δ is the step size between the different energy levels. It is worth noting that a reasonable upper limit for the energy levels is definable by considering the intensity values in the image. One possibility might be the determination of the minimal amount of energy required to reach the border of the selected ROI.

The higher the initial energy the more a ray can penetrate the high intensity region. If the region of the pupil is noise-free, rays with low initial energy reach the border of the pupil and stop at or close to the boundary points. If there is noise, some of the low energy rays stop before reaching the boundary, but most of the higher energy rays stop at the boundary.

For a given direction θ, for each initial energy $e_i \in E_I$ we can calculate endpoint ε^i_θ of the corresponding rays. The set of endpoints in a given direction θ will be denoted by $S_\theta = \{\varepsilon^i_\theta\}_{i=1,\ldots,n}$, where n is the number of rays that have been cast (the number of distinct initial energy levels). The propagation of rays in a homogeneous medium with a fixed linear step size difference in their initial energies leads to linearly spaced endpoints. In our case the structure of the eye is inhomogeneous, i.e. the pupil region's intensity is close to that of the vacuum, unlike the outer regions that mostly have higher attenuation. This fact causes a clustering in the endpoint positions. Overall, casting rays with monotonically growing initial energy leads to clusterings near the boundary points — specifically, near points with higher intensities. The main challenge is to properly select the correct clustering region that corresponds to the pupil boundary. We use hierarchical clustering to form the clusters. All the endpoints in a certain direction θ that are at a very small distance

from each other belong to the same cluster.

$$c_{\theta j} = \{\varepsilon_\theta^i \mid \text{maximal shortest distance between endpoints} < d\},$$

where d is a given small constant, $c_{\theta j}$ is a cluster of endpoints (where the distance between two properly selected endpoints is less than d). By using the initial assumption that beyond the pupil boundary there are mostly higher intensities, it is not hard to see that the rays with growing initial energy belong to the same cluster after reaching the pupil's border, because they can only penetrate a little bit into the higher intensity region, and their stop positions will be close to each other. Let $C_\theta = \{c_{\theta j}\}_{j=1,\ldots,m}$ denote the set of clusters for the direction θ, where m is the number of clusters. As C_θ is the clustered version of S_θ, $\bigcup_{j=1}^{m} c_{\theta j} = S_\theta$.

By using a set of initial energies with a higher density (i.e. smaller δ), it is easy to show that the cluster corresponding to the boundary region has the maximal cardinality.

Now let us find the cluster $c_{\theta k}$, where k is

$$k = \arg\max_j |c_{\theta j}|,$$

and $|c|$ is the cardinality of cluster c. In the selected cluster, the closest endpoint to the estimated pupil center (i.e. the endpoint with the smallest radial coordinate r) corresponds to the pupil boundary point $\beta_\theta = [\theta, r_{min}]$, where:

$$r_{min} = \min_i \|\varepsilon_\theta^i\| \ , \ \varepsilon_\theta^i \in c_{\theta k}.$$

For all directions parameterized by θ, collect the selected border points into a set denoted by $\mathbf{B} = \{\beta_\theta\}_\theta$, where θ is a set of directions; in our case $\theta = \{1, 2, ..., 359, 360\}$. This $\mathbf{B}$ is the output of our method, which is used later on to estimate the size of the pupil.

Figure 3 provides an illustration of he above-described process. The original image in Figure 3(a) has a good contrast but is noisy and blurry. For better visualization, only the pupil area is shown. We can follow the above-mentioned steps in a given direction. In Figure 3(d), the energy curves of rays with various initial energies are shown, and the square root of values are have been plotted to highlight lower initial energies. The curves are colored according to their endpoint cluster membership, which is shown in Figure 3(e). In Figure 3(g), the selected endpoints for all directions are indicated. Then in Figure 3(h), the result of filtering and ellipse fitting is shown. This last step will be discussed next.

5 Center point estimation and diameter measuring

Earlier, it was assumed that we have a good estimate for the pupil center point. In our approach, we use a geometrical relation between the center of circle and

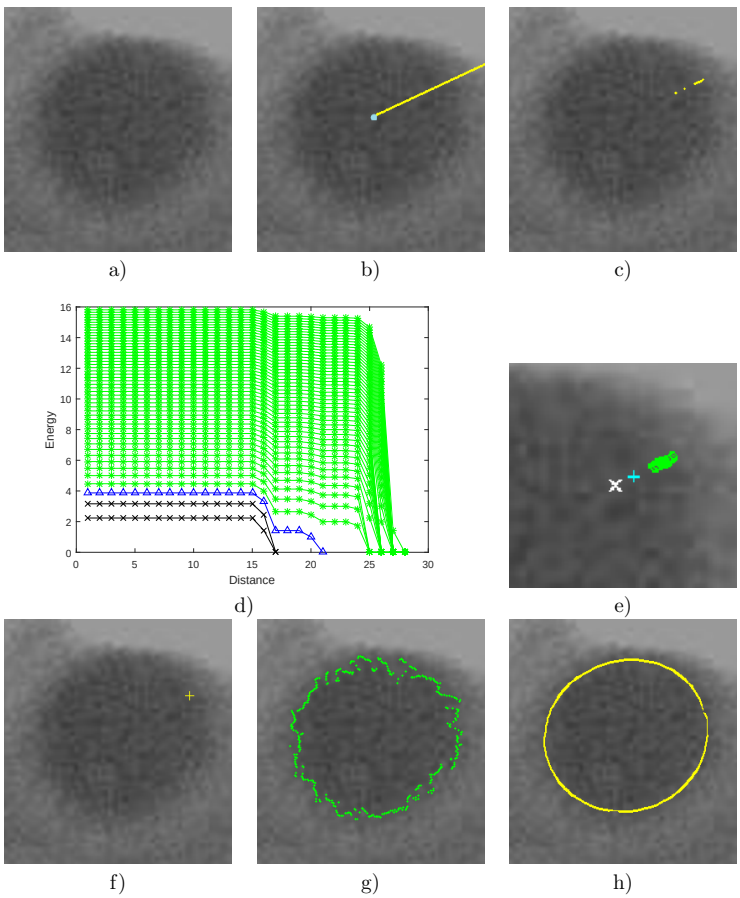


Figure 3: a) Original image, pupil area, b) selected direction of ray propagation, c) endpoints of rays, d) chart of ray energy curves at different initial energy levels, e) clustered endpoints, f) selected endpoint in the given direction, g) detected endpoints for all directions, h) the result of filtering and ellipse fitting.

its chords. We pick a point at random within the circle (Figure 4(a)) and draw chords through this point (Figure 4(b)). The perpendicular bisectors of the chords intersect at one single position, which is the center of the circle (Figure 4(c)).

The estimation process starts by placing seed points located in a grid. For each seed point having a pixel intensity near the minimal intensity in the image, we do the following. By using the energy attenuation model-based ray propagation method (Section 4), we cast rays from the actual seed point and find the most probable boundary points which give the endpoints of chords. After filtering the chords by their length, we calculate the perpendicular bisectors. It should be added that because of the inaccuracy of chord determination, the calculated lines

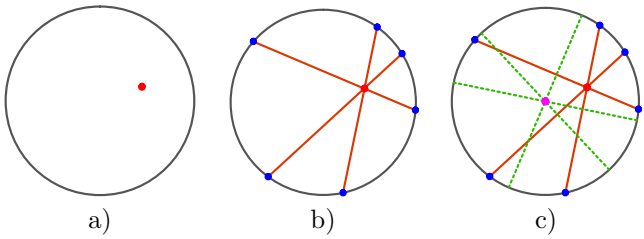


Figure 4: Circle center point estimation: a) random point within a circle, b) chords through the selected point, c) perpendicular bisectors and the center point of circle.

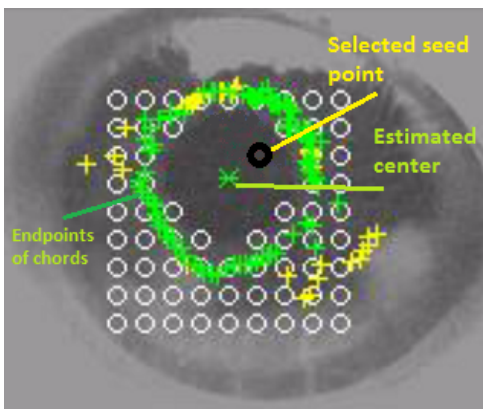


Figure 5: Estimation of the pupil center point.

do not intersect at a single point. To overcome this problem, we do a Least-Squares estimation to find the best intersection point for the given lines. Distances between the estimated center point and previously found boundary points are computed. The standard deviations of these distances then serve as scores for the corresponding estimated positions. After processing all the frames from the video, we have a huge number of pairs containing the estimated positions and its score. The output is the point that has the minimal score value. In Figure 5, the estimation process is shown. The seed points marked by white circles are filtered out at the beginning because their intensity was too high. The yellow plus signs mark the endpoints that have been filtered out because the corresponding chord length was too high compared to the median length of the others. It can be seen in the figure that from the selected seed point (black circle), the method estimated the center point (marked by a green star) with an acceptable accuracy.

After the selection of the best estimated center point, each frame is processed using this point as the source of radiation. The output for each frame is a set of endpoints and their distances, which may contain faulty detections due to high intensity reflections, undetectable boundary sections and occlusions. These factors

occur randomly compared to the regularity of pupil boundary points. Using this assumption, we will quantize the distances of endpoints and calculate their histogram. From the histogram, the distance with maximal magnitude is chosen since it is the most probable approximation of the pupil radius. Using this value, endpoint filtering is performed. Only endpoints with a distance close to the approximated radius are kept, on which Least-Squares based ellipse fitting is performed. Due to the relative position between the animal and the camera, the pupil is not necessarily a perfect circle; it may have a slightly elliptical shape. Anatomically the pupil is a circle whose radius is shortened in one direction in the video images due to the viewing angle. To produce the most realistic measure of the pupil diameter, we extract the longest axis from the fitted ellipse. If there is insufficient information (too few endpoints after filtering) to fit an ellipse, we just use the approximated radius length. The output of the automated process is the curve of the pupil's diameter change during the experiment, which is median filtered to eliminate bad measurements. Sample results of filtering, measurement and ellipse fitting phase are shown in Figure 6. In each picture, the green symbols represent the retained endpoints, while the red ones denote the outfiltered endpoints. The ellipses fitted to the retained endpoints are represented by yellow colored ellipses.

6 Evaluation and results

We evaluated our automated pupillometry on 20 videos, each containing 450 frames. The outcome of the automated process was then compared to results obtained by manual correction. We also developed a graphical user interface to support the correction of inaccuracies. After every fifth frame the user can correct the diameter length by hand if necessary. We did it this way, because the spectra of the intensity parameters is bigger than those having the same number of supervised measurements in consecutive frames. It should be added that in most cases the pupil boundary is not sufficiently sharp to place separating points exactly on it. Acceptable measurement means that a human would place the boundary point very close (within a few pixels) to the point representing the automatically computed position. We compared our result only for the 1800 supervised frames and we list the results of statistical analysis in Table 1. For each frame $f \in F_v$ in video $v \in V$, the relative percentage error (RPE) of the diameter measurement is

$$RPE_{vf} = 100\% \cdot \frac{|d_{vf} - \hat{d}_{vf}|}{|d_{vf}|},$$

where d_{vf} is the corrected diameter, $\hat{d}_{vf}$ is the estimated pupil diameter, F_v is the set of indices of supervised frames in video $v \in V$, $V = \{1, 2, ..., 20\}$. In Table 1, the rows correspond to the aggregation of RPE for all frames in a video, the columns correspond to the aggregation of the per video performances for each video in the test dataset. For instance, for video v, the mean relative percentage error μRPE

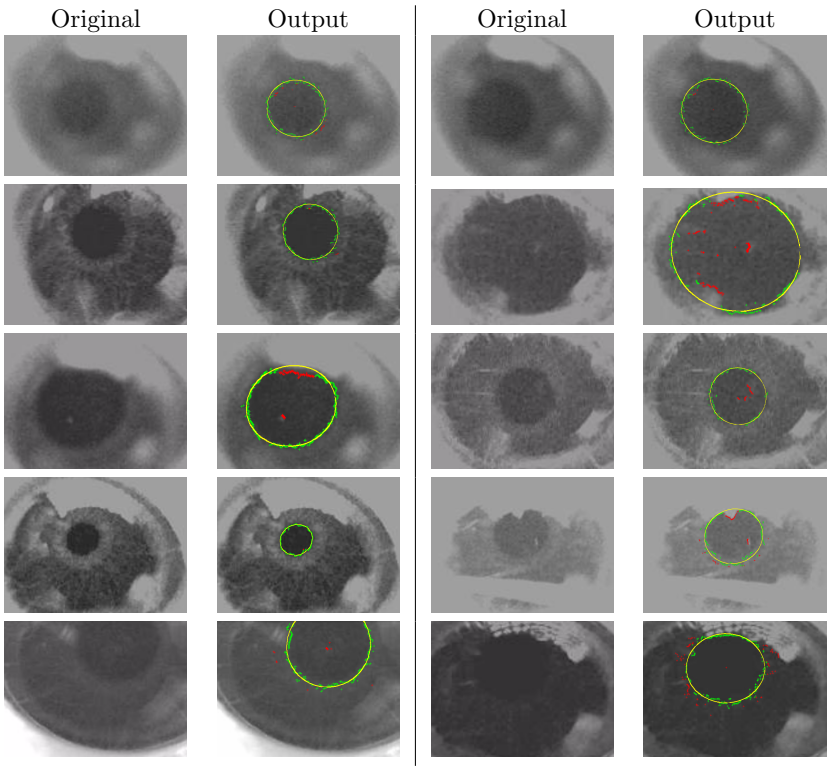


Figure 6: Results of endpoint filtering and ellipse fitting. The green symbols denote the retained endpoints, and the red ones denote the outfiltered endpoints. The yellow colored ellipses are fitted on the retained endpoints.

is just

$$\mu RPE = \frac{1}{|F_v|} \sum_{f \in F_v} RPE_{vf},$$

where $|F_v|$ is the number of supervised frames in video v. The maximum of the μRPE values in our case is 4.85% (which appears in the second data cell in the second row of Table 1). The upper part of Table 1 shows the results of analysis for all supervised frames, while the bottom part shows the same but only for the corrected frames. The last row gives information about the ratio of corrected frames.

As can be seen, for each supervised frame the mean relative percentage error is less than 2%, and in the corrected frames is around 4%, which is sufficiently accurate to assist medical staff in their work. On average, 40% of the frames were corrected by hand during supervision, but in most cases there were only minor differences between the estimated length and real diameter length. The proposed automatic method processes 3 frames per second, which includes ray tracking, filtering, hierarchical clustering, ellipse fitting and output video rendering.

Table 1: Performance analysis of the proposed method. The rows correspond to the aggregation of RPE for all frames in a video, while the columns correspond to the aggregation of the per video performances for each video in the test dataset.

		Minimum	Maximum	Mean	Standard deviation	Median
All supervised frames	Maximum relative error (%)	0.48	22.41	10.64	5.32	8.79
	Mean relative error (%)	0.60	4.85	1.95	1.38	1.53
	Standard deviation of relative error (%)	0.95	6.79	2.68	1.35	2.44
	Median of relative error (%)	0.00	3.10	0.90	1.23	0.00
Corrected frames	Mean relative error (%)	1.41	8.52	4.13	1.63	4.13
	Standard deviation of relative error (%)	0.98	7.19	2.94	1.58	2.46
	Median of relative error (%)	1.02	7.32	3.53	1.63	3.30
	Corrected frames (%)	15.28	88.89	46.38	23.03	40.97

Processing an input video takes around 3 minutes.

7 Conclusions

Here, we presented and described an automated process for measuring the pupil diameter in an infrared video. To accurately detect the boundary of the pupil, a novel energy attenuation-based ray propagation method was introduced. It can handle the wide spectra of intensity parameters, along with low contrast, blur and occlusions. The estimation of the pupil center point and robust diameter measuring was also outlined. We evaluated the proposed process on 20 videos and achieved an overall error rate below 2%. In the future, we would like to parallelize our computations and use more robust filtering of real boundary points. Also, it would be good to objectively compare our results with those produced by other similar methods.

References

[1] Ayala-Ramirez, Victor, Garcia-Capulin, Carlos H., Perez-Garcia, Arturo, and Sanchez-Yanez, Raul E. Circle detection on images using genetic algorithms. *Pattern Recognition Letters*, 27(6):652–657, 2006.

[2] Chen, Teh-Chuan and Chung, Kuo-Liang. An efficient randomized algorithm for detecting circles. *Computer Vision and Image Understanding*, 83(2):172–191, 2001.

[3] Cho, J.M., Lee, S.J., Kim, J.K., Choi, H.H., Kwon, O.S., and Kwon, J.W. A pupil center detection algorithms for partially-covered eye image. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume A, pages 183–186. IEEE, 2004.

[4] Cuevas, Erik, Ortega-Sánchez, Noé, Zaldivar, Daniel, and Pérez-Cisneros, Marco. Circle detection by harmony search optimization. *Journal of Intelligent & Robotic Systems*, 66(3):359–376, 2012.

[5] Cuevas, Erik, Osuna-Enciso, Valentn, Wario, Fernando, Zaldvar, Daniel, and Prez-Cisneros, Marco. Automatic multiple circle detection based on artificial immune systems. *Expert Systems with Applications*, 39(1):713–722, 2012.

[6] Cuevas, Erik, Sención-Echauri, Felipe, Zaldivar, Daniel, and Pérez-Cisneros, Marco. Multi-circle detection on images using artificial bee colony (ABC) optimization. *Soft Computing*, 16(2):281–296, 2012.

[7] Cuevas, Erik, Zaldivar, Daniel, Pérez-Cisneros, Marco, and Ramírez-Ortegón, Marte. Circle detection using discrete differential evolution optimization. *Pattern Analysis and Applications*, 14(1):93–107, 2011.

[8] Dasgupta, Sambarta, Das, Swagatam, Biswas, Arijit, and Abraham, Ajith. Automatic circle detection on digital images with an adaptive bacterial foraging algorithm. *Soft Computing*, 14(11):1151–1164, 2010.

[9] De Santis, A. and Iacoviello, D. Optimal segmentation of pupillometric images for estimating pupil shape parameters. *Computer Methods and Programs in Biomedicine*, 84(2–3):174–187, 2006.

[10] Illingworth, J. and Kittler, J. A survey of the Hough transform. *Computer Vision, Graphics, and Image Processing*, 44(1):87–116, 1988.

[11] Kim, Jieun and Park, Kyungmo. An image processing method for improved pupil size estimation accuracy. In *Engineering in Medicine and Biology Society, 2003. Proceedings of the 25th Annual International Conference of the IEEE*, volume 1, pages 720–723. IEEE, 2003.

[12] Lee, J.C., Kim, J.E., Park, K.M., and G., Khang. Evaluation of the methods for pupil size estimation: On the perspective of autonomic activity. In *Engineering in Medicine and Biology Society, 2004. IEMBS '04. 26th Annual International Conference of the IEEE*, volume 2, pages 1501–1504. IEEE, 2004.

[13] Miller, D.B., Benton, B.J., Hulet, S.W., Mioduszewski, R.J., Whalley, C.E., Carpin, J.C., and Thomson, S.A. An image analysis method for quantifying elliptical and partially obstructed pupil areas in response to chemical agent vapor exposure. In *Bioengineering Conference, 2003 IEEE 29th Annual, Proceedings of*, pages 63–64. IEEE, 2003.

[14] Rad, Ali Ajdari, Faez, Karim, and Qaragozlou, Navid. Fast circle detection using gradient pair vectors. In *Proc. VIIth Digital Image Computing: Techniques and Applications*, pages 879–888, 2003.

[15] Ritter, Nicola, Cooper, James, Owens, Robyn, and Van Saarloos, Paul P. Location of the pupil-iris border in slit-lamp images of the cornea. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, pages 740–745. IEEE, 1999.

[16] Tomasi, Carlo and Kanade, Takeo. *Detection and tracking of point features.* School of Computer Science, Carnegie Mellon Univ. Pittsburgh, 1991.

[17] Yuen, HK, Princen, J, Illingworth, J, and Kittler, J. Comparative study of Hough transform methods for circle finding. *Image and Vision Computing*, 8(1):71–77, 1990.

[18] Zhu, Danjie, Moore, Steven T., and Raphan, Theodore. Robust pupil center detection using a curvature algorithm. *Computer Methods and Programs in Biomedicine*, 59(3):145–157, 1999.

Acta Cybernetica **23** (2017) 615–628.

An Approach to the Quantitative Assessment of Retinal Layer Distortions and Subretinal Fluid in SD-OCT Images*

Melinda Katona[a] and László G. Nyúl[a]

Abstract

A modern tool for age-related macular degeneration (AMD) investigation is Optical Coherence Tomography (OCT), which can produce high resolution cross-sectional images of retinal layers. AMD is one of the most frequent reasons for blindness in economically developed countries. AMD means degeneration of the macula, which is responsible for central vision. Since AMD affects only this specific part of the retina, untreated patients lose their fine shape- and face recognition, reading ability, and central vision. Here, we deal with the automatic localization of subretinal fluid areas and also analyze retinal layers, since layer information can help to localize fluid regions. We present an algorithm that automatically delineates the two extremal retinal layers, successfully localizes subretinal fluid regions, and computes their extent. We present our results using a set of SD-OCT images. The quantitative information can also be visualized in an anatomical context for visual assessment.

Keywords: optical coherence tomography, SD-OCT, age-related macular degeneration, AMD, subretinal fluid, retinal layer segmentation

1 Introduction

Age-related macular degeneration is one of the most frequent reasons for blindness in economically developed countries. In the world, more and more people suffer from AMD, which presents a challenge to the health systems. AMD means degeneration of the macula which is responsible for central vision. Since AMD affects only this specific part of the retina, untreated patients lose their fine shape- and face recognition, reading ability, and central vision [11].

*This study was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP-4.2.2.D-15/1/KONV-2015-0024 'National Excellence Program'.

[a]Department of Image Processing and Computer Graphics, University of Szeged, E-mail: {mkatona,nyul}@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.11

In essence, AMD has two forms, namely dry and wet form, and the latter causes fast and serious visual impairment in 10% of the cases [14]. In this type of disease, abnormal angiogenesis starts from the choroid under the macula. Fluid and blood leak out of the neovascularized membrane to retina layers, and this ruins the photoreceptors.

Experiments have demonstrated that the vascular endothelial growth factor (VEGF) plays a vital role in the formation of choroidal neovascularization [5]. Currently, the most common and effective clinical treatment for wet AMD is anti-VEGF therapy, which is a periodic intravitreal (into the eye) injection [12].

In the last decade, Optical Coherence Tomography (OCT) has been widely used in the diagnosis of AMD and follow-up therapy. Spectral domain OCT (SD-OCT) produces 3D image volumes, which have been useful in clinical practice. Existing OCT systems are partially suited to monitoring the progress of the disease, but OCT reveals many features about AMD such as hyper-reflective dots (HRD), subretinal fluid and cysts. Figure 1 presents an SD-OCT B-scan with biomarkers of AMD.

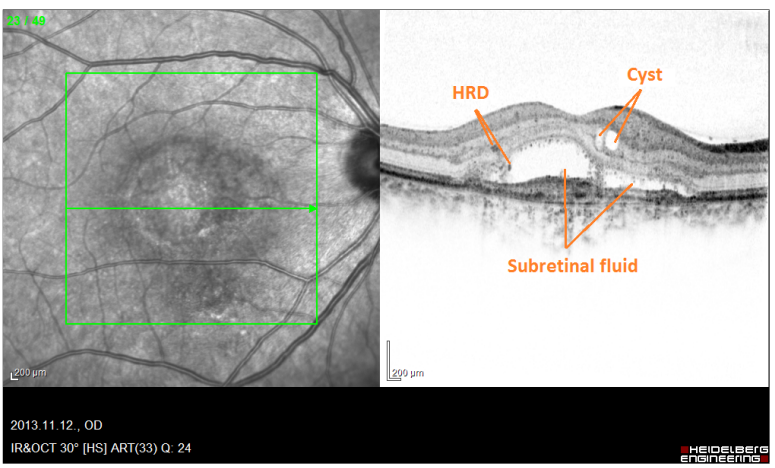


Figure 1: Original Optical Coherence Tomography (SD-OCT) image with biomarkers of AMD.

In the literature, a large number of publications exist on detecting retinal layers based on various techniques. One approach is the automatic segmentation procedure using graph theory [1, 2, 8]. In this approach, the graph nodes usually relate to image pixels, the graph edges are assigned to pairs of pixels, and the edge weights depend on the intensity differences between the two pixels, and also may depend on spatial distance between the pixels. Image segmentation then becomes a graph cutting problem, which can be handled by dynamic programming. These approaches are less tolerant to noise, which is a disadvantage, because images are often very noisy. Another basic idea relies on the well-known energy-minimizing active contour method which, unfortunately, cannot handle low contrast and noise.

Yazdanpanah et al. [21] suggested a multi-phase framework with a circular shape prior for modeling the boundaries of retinal layers and estimating the shape parameters. They used a contextual scheme to balance the weights of different terms in the energy functional. Also, machine learning has been widely used in recent years for retinal image analysis. Lang et al. [13] used a random forest classifier to segment retinal layers. The random forest classifier learns the boundary pixels between layers and produces an accurate probability map for each boundary, which is further processed to finalize boundaries. Procedures based on active contour or machine learning provide an effective solution, but these methods are too time-consuming. Hassan et al. [10] used a structure tensor approach combined with a nonlinear diffusion process for layer detection. A structure tensor is a second-moment matrix that displays similarities and prominent orientations of the image gradient. Some other approaches use optimized boundary tracking [7] or polynomial smoothing [15]. These algorithms are rather complex.

Many studies focus on automatic detection of subretinal fluid in OCT images. One of the most popular approaches is based on the Split-Bregman optimization technique [9]. This method is used to segment dark regions (depending on the image acquisition settings) between layers. These segmented regions are treated as possible fluid candidates. A random forest classifier is trained to distinguish true fluid regions from false segments [3, 22]. Also, a fuzzy level set method was introduced by Wang et al. [19] to identify fluid-filled regions. They use the combination of three types of scans (two types of B-scans and a C-scan) to generate a comprehensive volumetric segmentation of the retinal fluid. The remaining artifacts are removed by identifying morphological characteristics and vascular shadowing. Novosel et al. [16] recommended a locally-adaptive loosely-coupled level set method. This approach exploits the local attenuation coefficient differences of layers around an interface to delineate the fluid. This concept can also handle abrupt attenuation coefficient variations and topology-disrupting anomalies. SEAD (symptomatic exudate-associated derangements) segmentation in 3D volumes plays and important role in the treatment of neovascular AMD. The accurate detection is a challenge because of the large diversity of SEAD size, location and shape. Xu et al. [20] proposed a voxel classification-based approach using a layer-dependent stratified sampling strategy to address the class imbalance problem in SEAD detection.

In this paper, we deal with the automatic localization of subretinal fluid areas and also analyze retinal layers, since layer information can help to localize fluid regions. We present an algorithm that automatically delineates the ILM (inner-limiting membrane) and RPE (retinal pigment epithelium) retinal layers, successfully localizes subretinal fluid regions, and computes their extent. We present our results using a set of SD-OCT images and we depict our results in two different ways. Our proposed method is based on simple operations that can detect important regions quickly and efficiently. Each layer can be distorted by the effect of the disease. Our method can estimate the distortion rate and possible normal layer boundary, which may be useful for doctors. This feature of our algorithm makes it special, whereas the above-mentioned algorithms cannot estimate the boundary of a normal layer.

2 Methods

Now, we will present our proposed method for detecting boundary layers and subretinal fluid regions. The procedure first delineates the inner and outer boundary retinal layers (ILM and RPE, resp.) using vertical profiles of OCT cross-section images. Then subretinal fluid regions are localized and delineated. This is followed by calculating quantitative measures such as the extent of subretinal fluid in each slice, thickness and creasing of retinal layers. Once we have this information for each slice, other (regional or global) metrics (e.g. subretinal fluid volume) can also be readily computed.

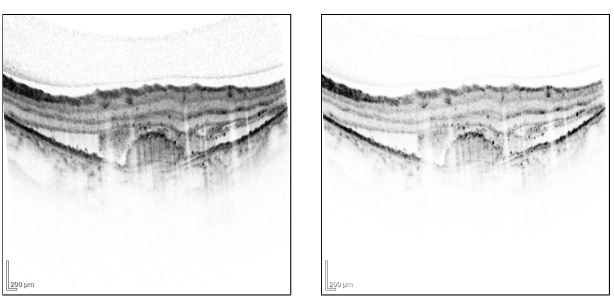
2.1 ILM and RPE layer extraction

The OCT images are affected by distortions like "shadowing" by blood vessels, and these may lead to false detections. First, we improve the image quality by noise filtering and contrast enhancement using a fuzzy operator [4]. This can highlight major retinal layers. We analyze vertical profiles of the filtered image and large intensity steps in pixel density are assumed to correspond to change in tissue. The function is defined by the expression

$$\kappa_\nu^* = \frac{1}{1 + \frac{1-\nu}{\nu}\left(\frac{1-x}{x}\frac{\nu}{1-\nu}\right)^\lambda}\,, \tag{1}$$

where ν is the threshold, x is the pixel intensity and λ denotes sharpness of the filtering. As we mentioned earlier, the κ_ν^* function can highlight boundary layers and help suppress noise. We determined dynamically the input parameter ν in a simple way. We sampled from the top range of the image and calculated the average intensity for this ROI. The λ parameter value was set to 3, empirically. Figure 2 shows an example where the κ_ν^* function was applied.

After filtering, we divided the image into bars with fixed width. A bar consisted of 10 consecutive pixel columns and we calculated the horizontal projections of each



(a) (b)

Figure 2: Sample OCT image before (a) and after (b) applying the κ_ν^* function.

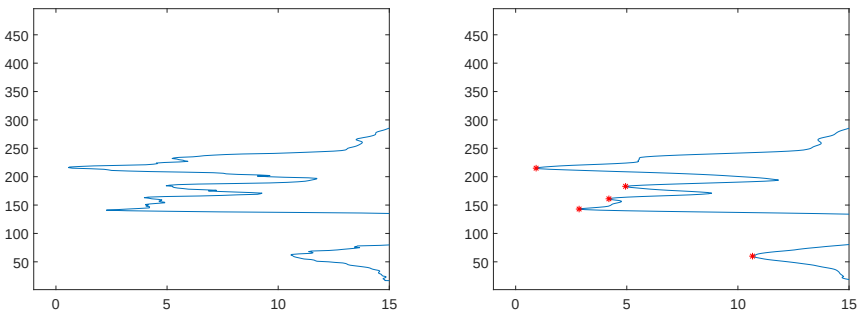


Figure 3: Calculated horizontal projection of a bar (a) and the Savitzky-Golay filtered data (b).

bar to determine the boundaries. One of the major steps of our proposed procedure is to analyze the vertical profiles. As this signal was noisy, it was necessary to filter the data. We used the Savitzky-Golay filter [18], which is a smoothing digital filter. This filter is good at preserving the relevant high frequency components of the signal, which is an important aspect for our detection method. In Fig. 3, there is an example of a projection (Fig. 3(a)) and its filtered version (Fig. 3(b)).

To determine of the outer layer boundary is harder than the inner boundary, because Choriocapillaris and Chorodoidal vessels are located under the RPE layer. The intensity of these regions vary, so several peaks appear in the projections. Fortunately, in most cases, these minimum points are not prominent, and do not cause a problem in the choice of the right locations. The algorithm chooses the outstanding local minimum from the projected data to identify the possible inner and outer layers.

Some of our OCT image volumes are strongly affected by noise or blood vessels shadows. To achieve a more efficient and reliable segmentation, we need to estimate the artifacts caused by the blood vessels. Below, we describe a method for detecting blood vessels from the region of interest (ROI), so we have information about the position of retinal blood vessels for each slice, which will aid layer boundary detection.

2.2 Blood vessel segmentation

Now, we summarize our simple approach for segmenting retinal blood vessels in fundus images. Fundus images are also provided with the OCT studies, in addition to the cross-section slices, and spatial correspondence is well defined. Hence we can use the fundus image for vessel segmentation and later use this information when processing the OCT slices.

Major vessels cause a bigger shadowing effect on the OCT slices than the minor

capillaries. Here, we do not seek to have a perfect segmentation of the whole vessel tree. The segmentation output will be used to identify those positions (bars) on the OCT slices where shadowing may be significant, since this can invalidate our underlying assumption about large intensity steps matching layer boundaries. Once we can localize these less reliable parts of the image, we can specially treat (or even exclude) these parts from the layer boundary detection algorithm and handle them as "missing data" rather than using "false information".

Many published methods in the literature [6] deal with the retina vessel segmentation problem and try to achieve the most accurate segmentation possible. For our purposes, a rough estimate is sufficient, as the vessels are not the objective of our studies, they merely locally spoil the contrast which hinders layer detection. It can be seen in Fig. 4(a) that the central region (the macula and its surrounding) is significantly darker than other parts of the image, hence the contrast between blood vessels and their surrounding background differs in the central and the peripherical parts of the image. Here our aim is to localize these regions and homogenize them in a simple way. For this, we use a Gaussian-pyramid technique with 4 levels. Figure 4(b)) depicts one level of the pyramid. Intensity homogenization is performed as a pixel operator according to the following formula:

$$I_{(x,y)} = C \cdot I_B(x,y)/I_O(x,y) \text{ for all pixels } (x,y) \text{ in the image }, \tag{2}$$

where I_O is the original image, I_B is the blurred image, and C is the maximal intensity in I_O. Figure 4(c) shows the result of applying (2).

Intensity homogenization is followed by a contrast enhancement step, using an adaptive fuzzy contrast stretching method, which is more effective than a commonly used contrast stretching procedure. Let I denote the input image, I_{max} the maximum intensity and I_{min} the minimum intensity of the image. The linear membership function $\mu_{i,j}$ is defined as

$$\mu_{i,j} = (I_{i,j} - I_{min})/(I_{max} - I_{min}) \,, \tag{3}$$

i.e., the membership value for pixel (i,j) corresponds to the degree of brightness of the gray level intensity of that pixel relative to the intensity range of the whole image. This is a simple way to assign fuzzy membership values to elements of a set (to the pixels, in our case). In a fuzzy processing approach, memberships are manipulated instead of original properties. We achieve contrast enhancement by using the intensification operator (INT) [17]

$$\mu'_{i,j} = \begin{cases} 2 \cdot (\mu_{i,j})^2, & \text{if } 0 \le \mu_{i,j} \le T \,, \\ 1 - 2 \cdot (1 - \mu_{i,j})^2, & \text{otherwise} \,, \end{cases} \tag{4}$$

where T is an adaptively calculated threshold value. We used the statistical mean of the intensities in each window to calculate T. Eq. (4) transforms membership values that are above the threshold to values that are much higher and membership values that are lower than the threshold to values that are much lower, in a nonlinear manner. The last step here is defuzzification, i.e., generating properties in the

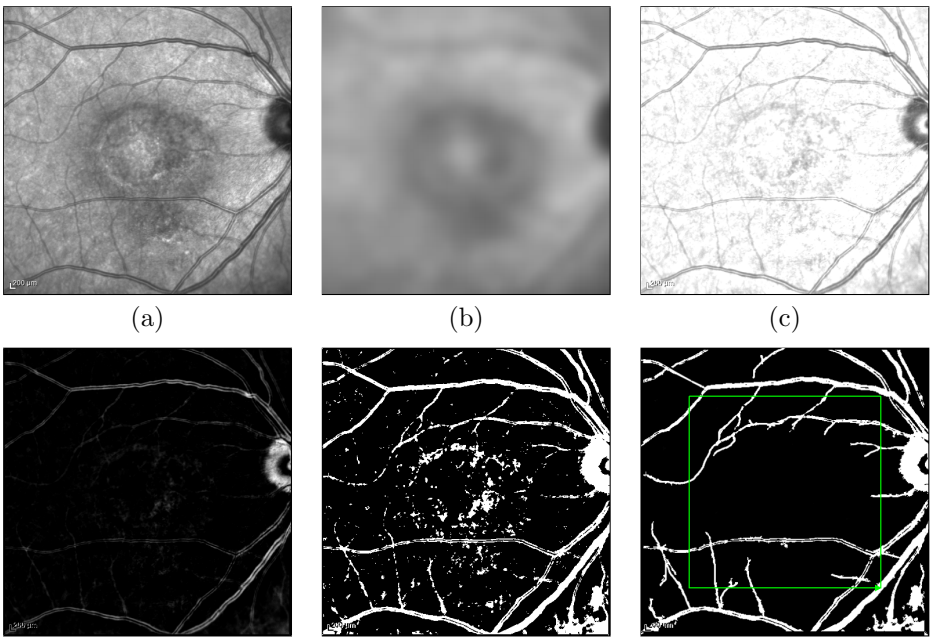


Figure 4: Main steps of retinal blood vessel segmentation: (a) original image, (b) one level of the Gaussian-pyramid, (c) homogenization, (d) fuzzy contrast stretching (in complementer image), (e) binarization, (f) output image (after removal of false objects).

original dimensions from the resulted memberships. $I'_{i,j}$ denotes the calculated new intensity for pixel (i, j), and it is got by applying the inverse of the transformation used for fuzzification:

$$I'_{i,j} = I_{min} + \mu'_{i,j} \cdot (I_{max} - I_{min}) \; . \tag{5}$$

Figure 4(d) illustrates the contrast enhanced image. The contrast enhanced image is turned into binary values in order to extract possible blood vessel regions, using adaptive thresholding in a sliding 15×15 window. In the next step, we eliminate false positive objects, because vessels constitute a connected object, so smaller segments perhaps arise from noise. We use morphological closing with a rectangle shaped structure element. The size of the element is consistent with the thinnest blood vessel. Afterwards, we reject all those regions whose area is less than 10 pixels and whose shape is not elongated. Once we have the vessel mask on the fundus image, we can project these data values onto the OCT cross-sections.

2.3 Automatic detection of subretinal fluid

Before turning to the subretinal fluid regions, we will assume that blood vessels and the ILM and RPE boundaries are already found in each OCT slice. Our procedure uses these data values when estimating subretinal fluid regions. Subretinal fluid is close to the RPE layer and appears as a larger hyporeflective connected component. For processing, we use the negative of signals, because our medical colleagues use the inverted presentation of images for the visual assessment and also exported the image data for us in this format. When layers are very creased due to AMD, these regions split up into multiple objects. To localize the fluid, we first use the Savitzky-Golay filtered signal that was introduced in Section 2.1. Once again, we process the OCT slices in vertical stripes, and based on the above assumptions, we look for minimum locations on the horizontal projections that are closest to the RPE layer. Some outliers can be filtered out based on assumptions about the layer thickness. Using the vessel shadow information, we can filter out those stripes that are less reliable than the others, and fit a smooth curve to the reliably detected minimum locations, thus approximating the fluid surface in the less reliable positions. In Section 2.1, we recommended a possible normal outer layer boundary. This can help us to define the degree of creasing of the layers and it also supports outlier filtering. These conditions reduce the dataset sufficiently, so after this step, we can fit a curve to the detected points and outline the subretinal fluid region.

3 Evaluation

3.1 Image data

Our evaluation dataset contained 11 Heidelberg Spectralis OCT scans of wet age-related macular degeneration patients treated with anti-VEGF intravitreal injections. The scanning parameters were: a 49 scan pattern, a pattern size of 5.8×5.8 mm, a distance between B-scans of 121 μm, a size X of 512 pixels, a size Z of 496 pixels; the pixel size was 11.44 μm and 3.87 μmm in X and Z directions, respectively.

Manual ILM and RPE layer segmentation was performed by ophthalmologists for 7 image sequences. This was treated as the ground truth for evaluating the boundary layer detection method on these 7 volumes.

3.2 Results and discussion

The proposed method was implemented in MATLAB, with the help of the Image Processing Toolbox. We evaluated our retinal layer detection algorithm in two different ways. Firstly, to compare the results of our algorithm against the manual delineations, we calculated the mean, maximum and standard deviation of boundary errors for every surface. The 7 curves shown in Figure 5 depict the error histogram for those OCT volumes where manual annotation was available. Each curve aggregates the boundary errors in the 49 scans (slices) of a study.

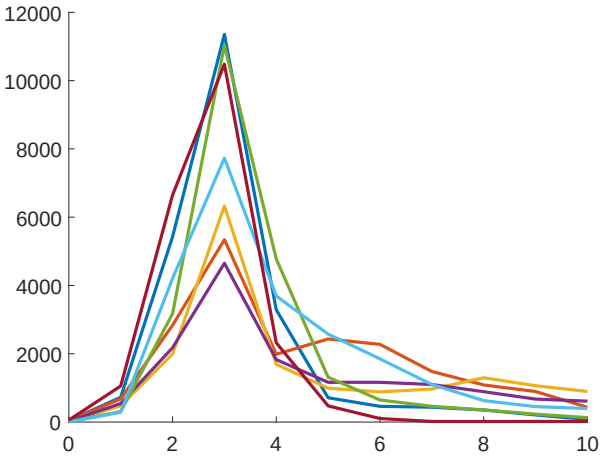


Figure 5: Error histogram of 7 image sequences.

Table 1: Summary of the mean, standard deviation, and maximum error (in pixels) between manually segmented and automatically detected layers in 7 annotated OCT image sequences.

	Mean	Standard deviation	Maximum
Seq_02	2.01	1.56	17
Seq_03	2.10	0.69	15
Seq_04	1.44	0.65	15
Seq_05	1.96	0.80	19
Seq_06	2.39	1.63	18
Seq_07	2.17	0.64	15
Seq_08	1.83	0.65	17
All	1.98	0.94	16.57

It shows that the highest error measure is between 1 and 4 pixels in most cases and Table 1 asserts to this statement. As shown in Table 1, the maximal distance between manually segmented and automatically detected layer boundary is 19 pixels (ca. 73.5 μm). This deflection comes from two sources: the substantial jumps between B-scans and layer distortions due to the disease. Unfortunately, we could not exploit 3D information directly to segment layers because there are some anomalies among slices of the OCT volume, due to the image acquisition and registration process (within the device's software).

Secondly, as you can see in the right-hand example in Figure 6, there is a big difference between the manually annotated and the automatically segmented outer

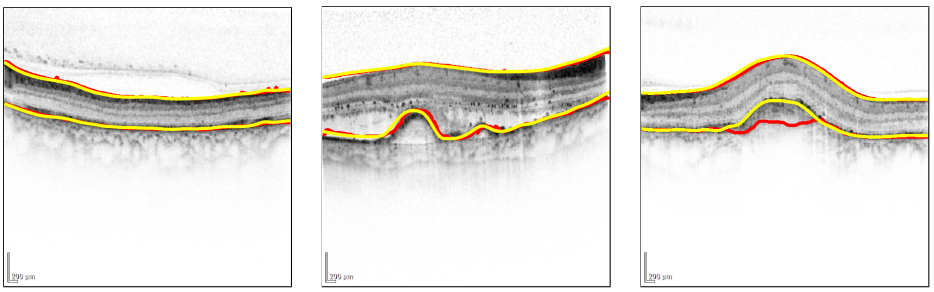


Figure 6: Visual comparison of expert annotated (in red) and automatically detected (in yellow) layers.

layer because our algorithm follows the distorted layers. Still, the recommended possible normal boundary aligns well with the manual annotation, so it avoids the false detection. Nevertheless, in most cases, the mean errors are less than 2 pixels and deviations are small between layers delineated by medical experts and the boundaries determined automatically, hence the difference is usually negligible.

In Section 2.3, we proposed a method for the automatic detection of subretinal fluid. Unfortunately, expert annotation was not yet available for the quantitative evaluation on a larger set of cases. Figure 6 illustrates the performance of the algorithm in a qualitative manner, presenting some example scans with overlaid manual layer segmentation.

The main advantage of our proposed method is its simplicity, i.e. it uses simple images processing operations which can be parallelized, and it does not need lots of parameters that are difficult to tune to the application (in contrast to the energy minimization approach, say).

Automatically calculated quantitative descriptors may be graphically presented to the reader to aid interpretation of data. The first is a traditional slice-by-slice display which provides a good depth context within a slice, but no spatial context between slices. In a colored overlay (image fusion), an anatomical display offers regional context and color encodes quantitative parameters. Figure 7 shows a restricted subretinal fluid area where the results were verified by ophthalmologists and they said that they the segmentation, quantification and also the visualization technique quite useful.

4 Concluding remarks

Here, we presented an algorithm for the detection of subretinal fluid areas and retinal layers and we presented some visualization techniques to illustrate the result. We calculated metrics to quantify features of the OCT from the perspective of AMD patients. After seeing the results, medical doctors at our clinic think that digital image processing can help in the quantitative assessment of the OCT features of AMD by providing automatic tools that can detect abnormalities and describe via

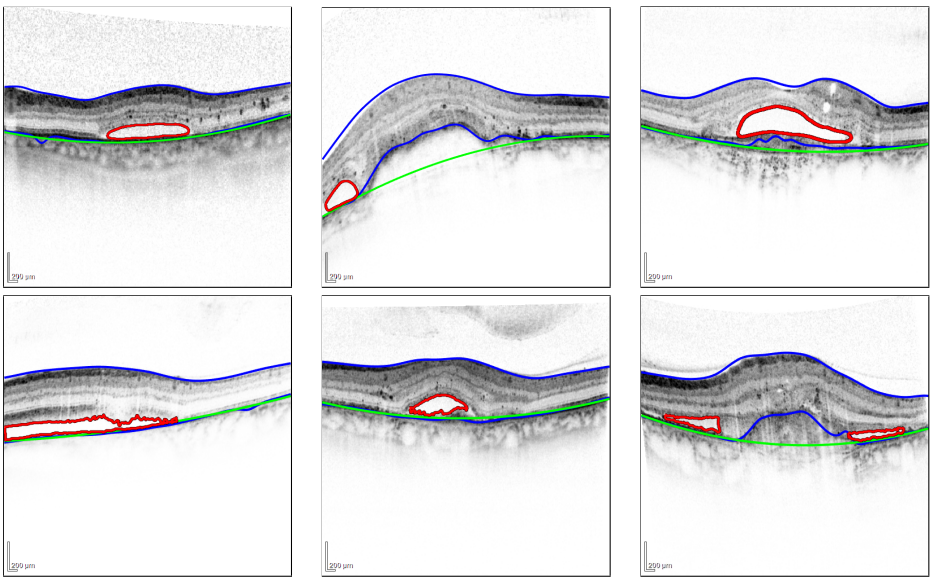


Figure 7: An illustration of detected boundaries and quantitation. The blue curves represent the detected boundaries, the green curve is the fitted normal layer boundary and the red curve shows the detected fluid volume boundary.

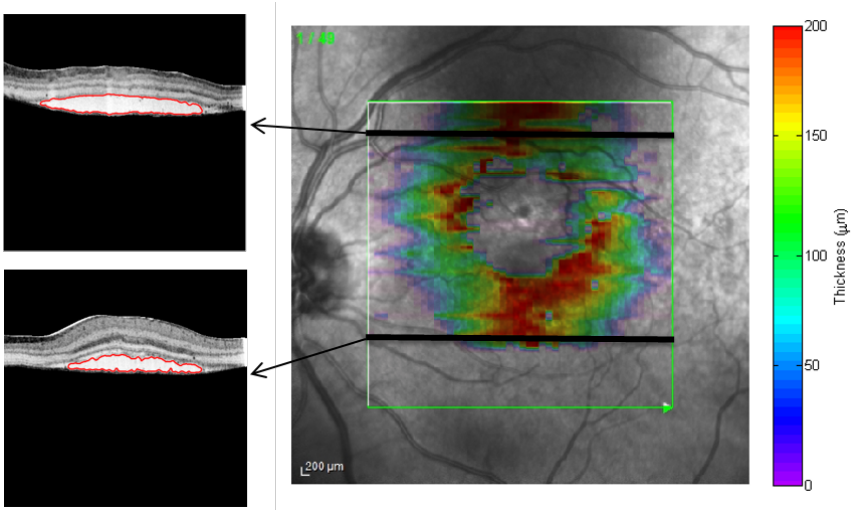


Figure 8: Color overlay of the subretinal fluid volume thickness. Left: The red boundaries indicate the subretinal fluid in each slice. Right: The color/hue represents subretinal fluid thickness.

objective metrics the current state of the disease and longitudinal changes during treatment. Using SD-OCT to follow up changes of subretinal fluid volume will become a useful tool for detecting subtle changes during the course of treatment. Further studies are planned in order to evaluate these new tools in a cohort of AMD patients.

Acknowledgements

The authors are grateful to Dr. Rózsa Dégi and Dr. Attila Kovács for providing the OCT data sets and for their medical advice. The authors would also like to thank Dr. József Dombi for suggesting the use of fuzzy operators in the image preprocessing step.

References

[1] Abhishek, A. M., Berendschot, T. T. J. M., Rao, S. V., and Dabir, S. Segmentation and analysis of retinal layers (ILM & RPE) in Optical Coherence Tomography images with Edema. In *Biomedical Engineering and Sciences (IECBES), 2014 IEEE Conference on*, pages 204–209, 2014.

[2] Chiu, S. J., Li, X. T., Nicholas, P., Toth, C. A., Izatt, J. A., and Farsiu, S. Automatic segmentation of seven retinal layers in SDOCT images congruent with expert manual segmentation. *Optics Express*, 18(18):19413–19428, 2010.

[3] Ding, W., Young, M., Bourgault, S., Lee, S., Albiani, D. A., Kirker, A. W., Forooghian, F., Sarunic, M. V., Merkur, A. B., and Beg, M. F. Automatic detection of subretinal fluid and sub-retinal pigment epithelium fluid in optical coherence tomography images. In *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 7388–7391, 2013.

[4] Dombi, J. *Modalities*, pages 53–65. Springer Berlin Heidelberg, 2012.

[5] Ferrara, N. Vascular endothelial growth factor: Basic science and clinical progress. *Endocrine Reviews*, 25(4):581–611, 2004.

[6] Fraz, M. M., Remagnino, P., Hoppe, A., Uyyanonvara, B., Rudnicka, A. R., Owen, C. G., and Barman, S. A. Blood vessel segmentation methodologies in retinal images a survey. *Computer Methods and Programs in Biomedicine*, 108(1):407–433, 2012.

[7] Fu, D., Tong, H., Luo, L., and Gao, F. Retinal automatic segmentation method based on prior information and optimized boundary tracking algorithm. In *Proc. SPIE*, volume 10033, pages 100331C:1–100331C:6, 2016.

[8] Garvin, M. K., Abramoff, M. D., Wu, X., Russell, S. R., Burns, T. L., and Sonka, M. Automated 3-D Intraretinal Layer Segmentation of Macular Spectral-Domain Optical Coherence Tomography Images. *IEEE Transactions on Medical Imaging*, 28(9):1436–1447, 2009.

[9] Goldstein, Tom, Bresson, Xavier, and Osher, Stanley. Geometric applications of the split Bregman method: Segmentation and surface reconstruction. *Journal of Scientific Computing*, 45(1):272–293, 2010.

[10] Hassan, B., Raja, G., Hassan, T., and Akram, M. U. Structure tensor based automated detection of macular edema and central serous retinopathy using optical coherence tomography images. *Journal of the Optical Society of America*, 33(4):455–463, 2016.

[11] Hee, M. R., Baumal, C. R., Puliafito, C. A., Duker, J. S., Reichel, E., Wilkins, J. R., Coker, J. G., Schuman, J. S., Swanson, E. A., and Fujimoto, J. G. Optical Coherence Tomography of Age-related Macular Degeneration and Choroidal Neovascularization. *Ophthalmology*, 103(8):1260–1270, 1996.

[12] Kovach, J. L., Schwartz, S. G., Jr., H. W. Flynn, and Scott, I. U. Anti-VEGF treatment strategies for wet AMD. *Journal of Ophthalmology*, 22:786870:1–786870:7, 2012.

[13] Lang, A., Carass, A., Hauser, M., Sotirchos, E. S.and Calabresi, P. A., Ying, H. S., and Prince, J. L. Retinal layer segmentation of macular OCT images using boundary classification. *Biomedical Optics Express*, 4(7):1133–1152, 2016.

[14] Lim, J.I. *Age-Related Macular Degeneration.* CRC Press, 2012.

[15] Lu, S., l. Cheung, C. Y., Liu, J., Lim, J. H., s. Leung, C. K., and Wong, T. Y. Automated layer segmentation of optical coherence tomography images. *IEEE Transactions on Biomedical Engineering*, 57(10):2605–2608, 2010.

[16] Novosel, J., Wang, Z., de Jong, H., van Velthoven, M., Vermeer, K. A., and van Vliet, L. J. Locally-adaptive loosely-coupled level sets for retinal layer and fluid segmentation in subjects with central serous retinopathy. In *2016 IEEE 13th International Symposium on Biomedical Imaging (ISBI)*, pages 702–705, 2016.

[17] Pal, S. K. and King, R. A. Image enhancement using smoothing with fuzzy sets. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(7):494–501, 1981.

[18] Schafer, R. W. What Is a Savitzky-Golay Filter? *IEEE Signal Processing Magazine*, 28(4):111–117, 2011.

[19] Wang, J., Zhang, M., Pechauer, A. D., Liu, L., Hwang, T. S., Wilson, D. J., Li, D., and Jia, Y. Automated volumetric segmentation of retinal fluid on optical coherence tomography. *Biomedical Optics Express*, 7(4):1577–1589, 2016.

[20] Xu, X., Lee, K., Zhang, L., Sonka, M., and Abrmoff, M. D. Stratified sampling voxel classification for segmentation of intraretinal and subretinal fluid in longitudinal clinical OCT data. *IEEE Transactions on Medical Imaging*, 34(7):1616–1623, 2015.

[21] Yazdanpanah, A., Hamarneh, G., Smith, B., and Sarunic, M. *Intra-retinal Layer Segmentation in Optical Coherence Tomography Using an Active Contour Approach*, pages 649–656. Springer Berlin Heidelberg, 2009.

[22] Zheng, Y., Sahni, J., C. Campa, A. N. Stangos, A. Raj, and Harding, S. P. Computerized Assessment of Intraretinal and Subretinal Fluid Regions in Spectral-Domain Optical Coherence Tomography Images of the Retina. *American Journal of Ophthalmology*, 155:277–286, 2012.

Acta Cybernetica **23** (2017) 629–643.

Phase Measurement Using DIC Microscopy*

Krisztian Koos[a], Begüm Peksel[a], and Lóránd Kelemen[a]

Abstract

The development of fluorescent probes and proteins has helped make light microscopy more popular by allowing the visualization of specific subcellular components, location and dynamics of biomolecules. However, it is not always feasible to label the cells as it may be phototoxic or perturb their functionalities. Label-free microscopy techniques allow us to work with live cells without perturbation and to evaluate morphological differences, which in turn can provide useful information for high-throughput assays. In this study, we use one of the most popular label-free techniques called differential interference contrast (DIC) microscopy to estimate the phase of cells and other nearly transparent objects and instantly estimate their height. DIC images provide detailed information about the optical path length (OPL) differences in the sample and they are visually similar to a gradient image. Our previous DIC reconstruction algorithm outputs an image where the values are proportional to the OPL (or implicitly the phase) of the sample. Although the reconstructed images are capable of describing cellular morphology and to a certain extent turn DIC into a quantitative technique, the actual OPL has to be computed from the input DIC image and the microscope calibration settings. Here we propose a computational method to measure the phase and approximate height of cells after microscope calibration, assuming a linear formation model. After a calibration step the phase of further samples can be determined when the refractive indices of the sample and the surrounding medium is known. The precision of the method is demonstrated on reconstructing the thickness of known objects and real cellular samples.

Keywords: DIC microscopy, image processing, phase imaging, quantitative phase microscopy

1 Introduction

In the area of optical microscopy, the simplest technique is brightfield microscopy, which was invented by Anton van Leeuwenhoek [30] in the 17th century as an

*This study was supported by the Hungarian National Brain Research Programme (MTA-SE-NAP B-BIOMAG), the CONCERT-Japan Photonic Manufacturing Joint Call, and by OTKA NN-114692.

[a]Hungarian Academy of Sciences, Biological Research Centre, E-mail: {koos.krisztian,begum.peksel,kelemen.lorand}@brc.mta.hu

DOI: 10.14232/actacyb.23.2.2017.12

improved version of the modern microscope developed by Robert Hooke [10]. Phase contrast microscopy was proposed in the early 1930s by Fritz Zernike, who was awarded the Nobel Prize in Physics in 1953 for his invention [25]. Phase contrast microscopy is interference-based and allows the study of the internal structure of living cells. Two years later, in 1955, Georges Nomarski established the theoretical basis for differential interference contrast (DIC) microscopy [24], which allows us to obtain information about the optical path length of the sample and shows features that are invisible in a brightfield microscope.

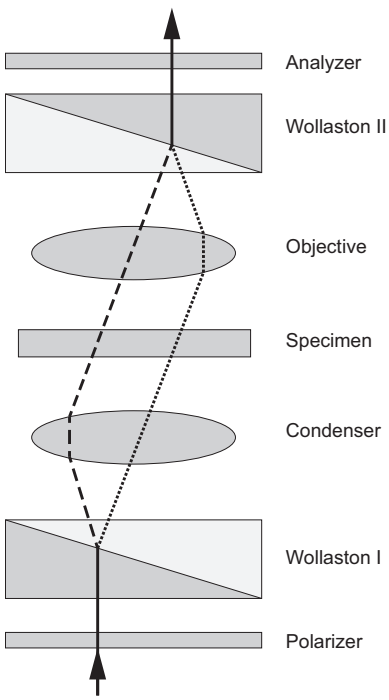


Figure 1: Setup of a DIC microscope. The arrows show the direction of the light. The dashed and dotted lines indicate the two mutually perpendicularly polarized light rays. The image formation starts as a light ray enters a polarizer that creates plane polarized light. A Wollaston prism splits the light into two perpendicularly polarized light rays, which are focused on the specimen by the condenser lens. These two light rays pass the sample by a minute shear. Materials with different refractive indices and the specimen's thickness generate a phase shift between the light rays. Next, a second lens, i.e. the objective, transmits the light to another Wollaston prism that recombines them. Finally, the analyzer creates plane polarized light perpendicular to the light of the first polarizer. The two light rays interfere at this point and generate a contrast image.

Brightfield, phase contrast and DIC microscopy are a set of of label-free techniques. Such techniques cannot be used to selectively visualize subcellular components, processes, or the localization of proteins, and this stimulated a demand for new methods in microscopy. In 1994 a discovery brought fluorescence to the forefront [32]. Proteins can be stained by fluorescent substances which show their localization inside the cells. Fluorescent images are quantitative, thus they may be analyzed by a suitable software. In contrast, label-free techniques, especially phase contrast and DIC, are qualitative methods. Although these images are easy to interpret with a human eye, conventional algorithms cannot be applied when it comes to image processing. However, DIC has some advantages over fluorescent microscopy: namely, cells can be observed without staining, so the technique is non-phototoxic and there is no need to fix the sample - in contrast to numerous fluorescent staining protocols. Phototoxicity often occurs upon the exposure of fluorescently labeled cells to light. Fluorescent molecules excited by light react with molecular oxygen that can damage subcellular components or the entire cell. Thus DIC allows live cell analysis in a physiologically more relevant way.

Over time, many techniques have been developed that convert a qualitative label-free microscopy into a quantitative technique. These techniques are collectively called Quantitative Phase Microscopy (QPM) [7, 18, 19]. An example is diffraction phase microscopy (DPM) [4, 12] which shifts the light all over the image and after a reconstruction step the phase differences are visible with high precision. However, the technique requires a special device and it is not widely used. DIC is a cost-effective solution and it should be turned into a quantitative technique. Determining the image formation model [11, 20, 21, 22, 28] of DIC is important in developing a correct reconstruction algorithm. The image in a DIC microscope is formed by using plane polarized light, separated by a Wollaston or Nomarski prism. The light rays are separated at a subpixel distance. The sample as a phase object shifts the light rays, which are then recombined and the very small differences produce a contrast image. The schematic DIC setup is illustrated in Figure 1 and Figure 2 shows an example DIC image.

Many mathematical models have been developed for the DIC image formation, some of which are listed below. The light in a DIC system is partially coherent and so it is not well defined. The level of coherence may vary inside the system which should also be taken into account. Moreover, the samples are usually not purely phase objects, hence the image is formed by both the phase difference and amplitude change. Theoretical developments were also carried out to reconstruct the amplitude along with the phase using the Transport of Intensity theorem [16, 33, 34]. DIC was converted into a quantitative technique earlier by modifying it to Phase-shifting DIC (PS-DIC) [6, 13]. PS-DIC utilizes an additional part that is able to introduce a constant phase shift to the light rays. PS-DIC virtually has the effect of rotating the sample – in other words, it changes the shear direction of the microscope, thus allowing edges otherwise invisible to be observed by the technique. Reconstructing the scene from multiple images using a linear model provides a result with a 15% error [3, 27]. However, physically rotating the sample is not possible after we have inserted the slide and using more images for a single

scene reconstruction increases the computational complexity and acquisition time. Attempts were made to reconstruct the true phase from single images using different reconstruction algorithms [31], but the results have not been validated. In this paper, we present a method that reliably converts DIC into a quantitative technique using just a single image.

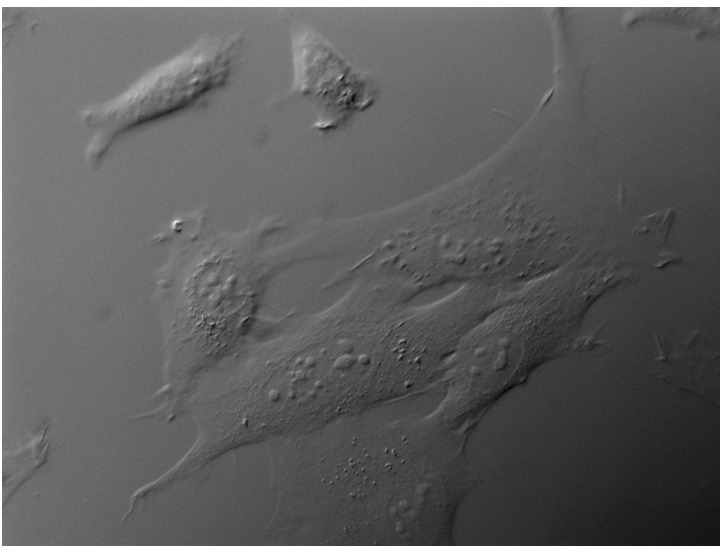


Figure 2: An example DIC image of cells. A well-configured DIC microscope provides sharp and detailed images. A trained eye can easily see the cells and the subcellular components, e.g. nucleoli. However, it is not straightforward to process, e.g. segment DIC images, without performing some transformation.

2 Methods

2.1 DIC reconstruction task

Images are treated as real-valued intensity functions parameterized by the image coordinates. Image coordinates and the domain of the intensity function are denoted by x, y and Ω ($\{x, y\} \in \Omega$). Local integration domains are small square windows W around the image points. In these local windows, parameters ξ and η are used to define the local coordinates aligned with the x and y coordinates of the image. The symmetric domain of the local windows have sizes δ in each direction: $\xi, \eta \in W$, $W = [-\delta, \delta] \times [-\delta, \delta]$. $d\Omega = dxdy$ and $dW = d\xi d\eta$ represent the infinitesimal area elements of the integrals in the image and in the local domains, respectively. Partial derivatives of nth order are denoted by ∂_x^n, ∂_y^n , and for $n = 1$ we use the notation ∂_x, ∂_y.

The aim of the DIC reconstruction is to create the primary image $I(x, y)$ based on the known DIC image $G(x, y)$, where parameters x and y are the pixel coordinates of the images.

DIC images are often treated as directional intensity difference images. This supposes that the image formation model is linear, which was previously shown to be highly correlated to advanced image formation models [15] in typical cases where the amount of diffraction is low. The direction is defined by the imaging device and given a priori. Mathematically, the reconstruction based on this simple interpretation can be formalized in the following way: we wish to retrieve the primary image I from its known directional difference image G. The two images are related by the equation

$$G(x) = I(x + d) - I(x - d), \tag{1}$$

where x is the parameter along the predefined direction and device parameter d is proportional to the phase shift difference between two light beams (reference beam and measuring beam). The difference on the right-hand side can also be represented by the integral $\int_{x-d}^{x+d} I'(\zeta)\, d\zeta$, where I' is the directional derivative of the primary image. The direct generalization of this integral is $\int_{x-d}^{x+d} K(\zeta - x)\, I'(\zeta)\, d\zeta = \int_{x-d}^{x+d} K(\zeta)\, I'(x + \zeta)\, d\zeta$ with some appropriately chosen finite kernel K. A further generalization to two dimensions can be defined by using the partial derivatives of the local integral

$$\hat{I}(x, y) = \iint_W K(\xi, \eta)\, I(x + \xi, y + \eta)\, dW, \tag{2}$$

with a two-dimensional finite kernel K such that

$$G = \mathbf{u} \cdot \nabla \hat{I}, \tag{3}$$

where $\mathbf{u} = \begin{bmatrix} u & v \end{bmatrix}^T$ is a unit vector ($u^2 + v^2 = 1$) in the shear direction determined by the imaging device. Note that image $\hat{I}$ is the convolved version of I and the integral (2) is a parametric integral w.r.t. the image coordinates x and y.

2.2 Reconstruction using a variational framework

We use a variational framework based on (3) to reconstruct DIC images. In a variational framework [5, 23, 26], we have to define an energy function that usually consists of a data term and one (or more) regularization term:

$$E = E_{data} + \lambda E_{reg}. \tag{4}$$

The local integral in (2) can be used as a data term in an energy functional expressing that the proper reconstruction of the primary image satisfies (3), for example

$$E_{data} \doteq \frac{1}{2} \iint\limits_{\Omega} \left[\mathbf{u} \cdot \nabla \hat{I} - (G - c) \right]^2 d\Omega \doteq \frac{1}{2} \iint\limits_{\Omega} S^2 d\Omega$$

$$S = \iint\limits_{\xi,\eta} K \left(u \partial_x I + v \partial_y I \right) d\xi d\eta - (G - c) \,. \tag{5}$$

Note that any arbitrary constant c can be added to the DIC image G without affecting the solution. The spatial regularization term ensures that we have locally smooth regions where G is smooth (i.e. there are no edges in the original image) and it has no effect on high image gradients. For this latter term, we use the total variation (TV) [29], which has the form

$$|\nabla I| = \sqrt{(\partial_x I)^2 + (\partial_y I)^2}. \tag{6}$$

To apply the gradient descent algorithm, the Euler-Lagrangian (EL) of the two terms in (5) and (6) have to be calculated. The complete derivation of the EL equations can be found in [14] and [15], but here we only describe and explain the final equations. The EL equation of the data term is very complex because of the local integrals introduced by the kernel function. Instead, we calculate the Taylor series of the perturbation function. Since the light ray pairs, separated by the DIC prism in the microscope setup, are at a subpixel distance from each other, we assume the perturbation function to be a piecewise analytic function, which is a requirement before replacing it by the Taylor series. In other words, the Taylor series is convergent around each image point, or at least in the domain of the local integration. In practice, we found that the first two terms provide sufficiently accurate results.

At this point, the gradient descent algorithm can already be applied iteratively. However, the intuition that the kernel function (or point spread function, PSF) has to encode some directional derivative effect is not straightforward here. Currently, the shadow-cast effect is described by the $u \cdot \nabla$ operator and although K can be chosen arbitrarily for the equation to remain true, an obvious choice would be a rotationally symmetric function K_R. With a parameter transformation, the calculation of derivatives can be moved from the image to the kernel, which allows the use of an approximation of the DIC PSF. This way, the second order derivatives of the PSF function have to be calculated and then used as a convolution kernel, which is computationally more expensive than calculating image derivatives and convolving with a single kernel. The rotationally symmetric function K_R further reduces the 'bilinear' equation to the following 'piecewise constant' approximation:

$$0 \;=\; \mathbf{u} \cdot \nabla G - \iint\limits_{\xi,\eta} K_R \mathbf{u} \cdot \nabla \nabla I \cdot \mathbf{u} d\xi d\eta. \tag{7}$$

The EL equation of the TV term has the following form [1, 29]:

$$0 = \frac{\delta}{\delta x}\left(\frac{\delta_x I}{|\nabla I|}\right) + \frac{\delta}{\delta y}\left(\frac{\delta_y I}{|\nabla I|}\right) \tag{8}$$

It is known that TV can remove noise, while preserving sharp signals.

2.3 Other reconstruction algorithms

In our recent paper [15], we described and compared the most relevant DIC reconstruction algorithms. These algorithms can be organized into groups based on their computational principles.

One major group is based on inverse filtering. Inverse filtering algorithms describe the reconstruction problem as a division in Fourier space. Frequency space calculations are very fast and these algorithms are direct, i.e. they provide a solution in one step. The Hilbert transform [2, 9] is considered to be the first algorithm that was used to reconstruct DIC images. Another widely used algorithm is Wiener filtering [9, 31], which avoids division by values close to zero. A third approach from this group is based on pure Fourier space operations with additional regularization terms [35] (referred to as Yin). Regularization terms in frequency space also help to avoid division by zero and they help to remove artefacts from the image and provide a smooth reconstruction.

A second group of algorithms is based on linear programming. The reconstruction problem is formed as a linear equation system, which is then solved by some algorithm. The image formation model is coded in a transfer matrix and the regularization terms, i.e. smoothness or sparsity are also expressed in matrix forms. Two linear equation system solvers have been proposed [17] to solve the DIC reconstruction problem, namely the sparseness-enhanced multiplicative update (SEMU) and a second order cone program (SOCP).

The third group of DIC reconstruction algorithms use a variational framework. The algorithm presented in Section 2.2 falls in this group (referred to as Koos), while another energy minimization method was presented in [8] (Feineigle). The Feineigle algorithm considers image formation as a directional derivation and does not incorporate the PSF of the microscope system, and the energy function is minimized based on its Eulerian equation.

Figure 3 shows 3 sample image series to compare the above mentioned algorithms. The input simulated DIC images are computed from the ground truth images by convolving them with a PSF, assuming a linear image formation model, which is known to be very close to the correct image formation model [15]. The difference in the output quality of the algorithms is visible. Better performing algorithms produce correct, visually indistinguishable results from the ground truths. The reconstructions of weaker algorithms, usually the inverse filtering based ones, are either blurred or only the outlines of the objects are correct. Furthermore, the algorithms often fail to reconstruct small changes, e.g. gradients in the ground truth, which is a necessity when it comes to cells. This effect is shown in the third,

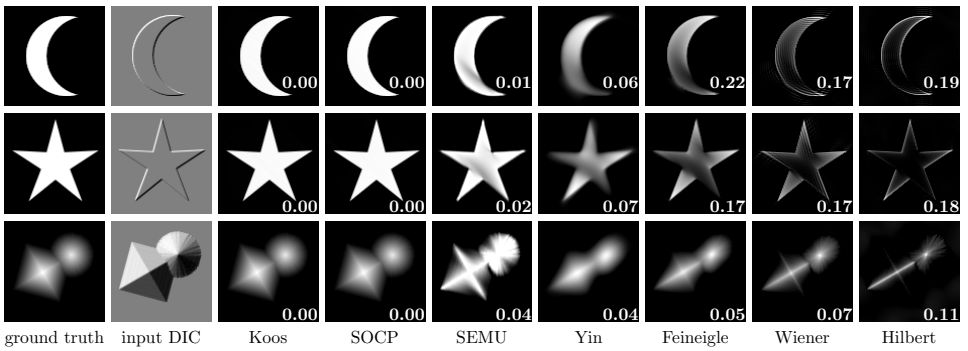


Figure 3: A comparison of DIC reconstruction algorithms. The first column contains ground truth images that the algorithms attempt to reconstruct. The images in the second column are the simulated DIC images of the ground truths, which are the input data of the algorithms. The shear direction in the simulation is -45 degrees, which is passed as prior knowledge to the algorithms. The following columns contain result images in decreasing order based on the algorithms' quality measured by the averaged mean squared error (MSE) taken from 20 images and their rotations. The values in the reconstructed images are the MSE values.

grayscale example in Figure 3. Since the variational framework presented in Section 2.2 performed the best in our comparison study, we decided to apply it in all subsequent computations.

2.4 An algorithm for phase and height reconstruction

In this section, we describe an algorithm used to measure the phase and height of unknown objects. The algorithm starts with a calibration process that aims to determine the relation between the values in the reconstructed images and the phase of the examined objects. The assumption of a linear image formation model has to be made, because it is already required by the reconstruction algorithms. The linear image formation model (in terms of phase) means that the sample is a phase object and does not absorb light, i.e. the amplitude of light does not change. Cells only absorb about 10% of light and in practice they are considered as phase objects.

As a first step, the microscope settings have to be fixed: the illumination and exposure time have to be set to a constant value, and a bandpass filter should be used if available. Using filters makes it easier to calculate a weighted average of wavelengths or a narrower range can be used later to calculate the phase. Furthermore, any other microscope specific parameters should be fixed that affect the image acquisition, i.e. image histogram operations should be disabled. Most microscope software can automatically determine these parameters, but only for the

sample being observed. The final settings can be based on the automatic values but they should be set manually so that they are suitable for both the calibration objects and the further samples.

The next step is the phase calibration. Images should be taken of some calibration samples. Calibration objects usually have simple shapes and are homogeneous. Moreover, the refractive index of both the sample and the surrounding medium has to be known in advance. After reconstructing the DIC images of the calibration samples, the reconstructed object values are fit to the theoretical phase profile. Now the function that maps the reconstructed values to phase values can be calculated. This function can be used later (without altering the microscope settings) to determining the phase of unknown objects, but with known refractive indices. If the objects are not homogeneous, an averaged refractive index value can be used to get an approximation, as will be shown later in Section 3.2.

To perform the phase calibration, we used polystyrene microbeads of diameter $d = 9\mu m$ (Polybead, Polysciences, Warrington, PA, USA). The refractive index of the beads is $n_{bead} = 1.595$ and $n_{oil} = 1.515$ for the oil (type DF, Cargille, Cedar Grove, NJ, USA) in which they are immersed. We used an Olympus IF550 bandpass filter for our tests which has an average weighted wavelength λ of about 550 nm. This λ value was then used to calculate the theoretical phase profile of the beads. The following equation gives the phase difference between the surrounding medium and the center of the bead:

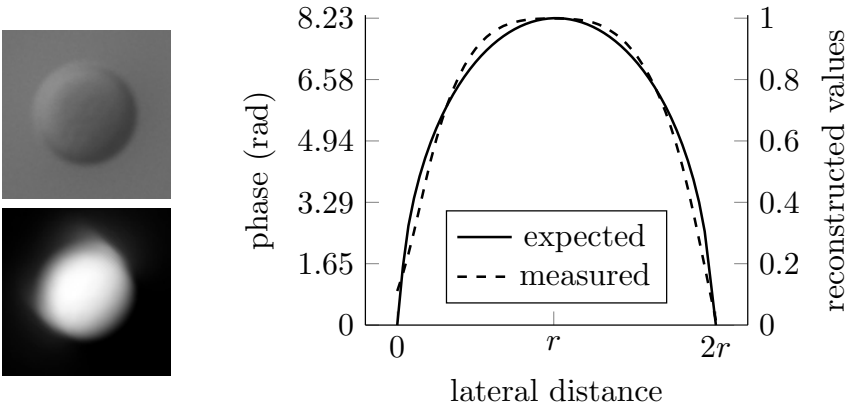
$$\varphi = \frac{2\pi}{\lambda}\Delta n d, \tag{9}$$

where $\Delta n = n_{bead} - n_{oil}$ in this case. Figure 2.4 shows an example DIC and reconstructed image of a microbead, and the averaged data fit to the theoretical phase profile. The data was normalized to the theoretical profile, which is a reliable method in the case of spheres, because the phase distribution is not uniform. The two line plots highly correlate. The flat-top behaviour of the measured data is attributed to the slight smoothing effect of the TV term, which preserves high jumps and flattens small differences. Based on the scales shown for the two y axes, a unit change in pixel intensity can be expressed as a phase change. This value can be used to express the phase of unknown objects. However, this approach does not consider diffraction effects due to the linearity of the model. If the phase difference is too high between the surrounding medium and the calibration object, diffraction causes artifacts in the DIC image which negatively affects the calibration.

The physical height of homogeneous samples can easily be calculated from the phase φ, if we know the wavelength λ and the refractive index difference Δn of the sample and the medium, using the following formula, which gives the height in nanometers:

$$h = \frac{\varphi\lambda}{2\pi\Delta n} \tag{10}$$

Algorithm 1 summarizes the process used for calibration and height estimation:



(a) (b)

Figure 4: Phase calibration on microbeads. (a) Example DIC image of a microbead and its reconstruction. (b) Line profile of the expected phase distribution of a microbead and that of the reconstructions, where r is the radius of a microbead. The plot of the reconstruction was averaged over 16 beads after normalization, in 4 directions (horizontal, vertical and two diagonals) through the center of the beads.

Algorithm 1 Measuring phase using DIC microscopy

1: Select a calibration sample with known dimensions and refractive index
2: Perform a phase calibration
3: Create images of the sample
4: Reconstruct the DIC images
5: Transform the reconstructed images to phase images
6: Optionally convert phase to height

If more than one type of sample needs to be imaged in one session, the calibration only has to be performed once, and from the second type the algorithm can be started from Step 3.

3 Results

In this section the algorithm described in Section 2.4 is validated on microbeads of different sizes. Then an example application is shown on images of cells, where the height of cells have also been calculated.

3.1 Phase reconstruction on microbeads

To verify that the calibration is accurate, we created images of microbeads of different but known sizes under the same conditions, in the same medium. Then

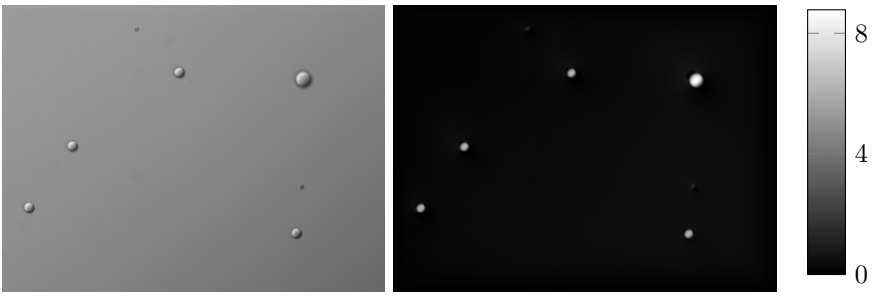


Figure 5: Example DIC image (left) of multiple sized microbeads and its reconstructed phase image (right) with a colormap. The phase was calculated after the phase calibration process (described in Section 2.4) had been performed.

the images were reconstructed and converted to phase images using the calibration function that was determined in the calibration process in Section 2.4. Finally, the observed peak values of the beads were compared to the theoretical peaks.

The diameters of the microbeads used for this test were 6 and 9 μm. The microbeads, originally supplied in distilled water, were mixed and put on a coverslip. Then the coverslip was dried under heat to ensure that the microbeads were not drawn to each other when the water evaporated, thus reconstructed images should be more accurate. 7 DIC images were made of the coverslip and the peak values of 29 smaller and 7 bigger microbeads were determined. The average reconstructed phase for the 6 μm beads were 5.81 rad, which is 0.33 rad more than the value that (9) gives. However, the variance in 6 and 9 μm beads were only 0.12 and 0.14 rad, respectively. We think that the phase error between the different sized beads originates from the nonlinearity of the image formation and it is not a problem of the reconstruction process, because the variance is much less among the same sized beads. Furthermore, 0.33 rad converted to distance is about 180 nm, which is fairly below the 10% size variance of the beads as given by the manufacturer. Figure 5 shows an example DIC image of microbeads and its reconstructed phase image.

3.2 Cell height measurement

To demonstrate the practical utility of phase reconstruction from DIC images, pictures of mouse embryonic fibroblast (MEF) cells were taken and reconstructed. Figure 6 contains example DIC images and their computed phase images. Rounded-up cells are brighter, while elongated (deformed) ones that have become attached to the bottom of the dish are mid-gray. Rounded-up cells are thought to have a similar volume to others, but they are taller. Since OPL is formed by the product of refractive index and thickness, taller cells appear brighter in the reconstructions. The nucleoli and other subcellular compartments appear as regions which demonstrates the resolution capability of the technique. Since (10) is linear, in the case of cells this transformation usually does not reflect the true height value. Although

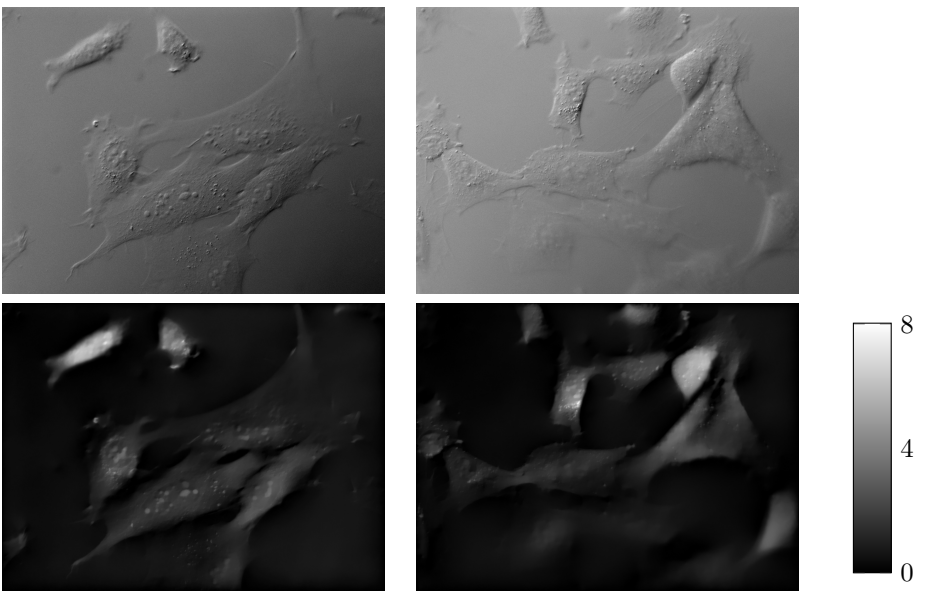


Figure 6: Example DIC images of MEF cells and their reconstructed phase images. The colorbar is in radians. The phases were calculated after the calibration process. Rounded-up cells are taller, hence appear brighter in the reconstructed images. The nucleoli and other compartments of the cells are clearly visible.

the nucleoli in Figure 6 have higher OPL, this is due to their higher refractive index and the cell height being smooth. If height information of cells is required, some statistical phase value (e.g. mean or median) should be used in the calculation. Furthermore, note that in spite of the microbeads having similar radian values in Figure 6 to Figure 5, it does not imply that the cells are of similar size to the microbeads. The refractive index difference Δn has a different value in the two cases, which results in significantly different height values when (10) is applied. The calculated height is about 10 μm and 20 μm, respectively, for attached and rounded-up cells which is in the range of the physical cell size.

4 Discussion

In this paper, a computational method was described to measure the phase of near-transparent objects using DIC microscopy. The method is based on a calibration process using an object with known dimensions and refractive index. The DIC images are reconstructed using a variational framework. The reconstructed images are then converted to phase or height images. The process is described in a step-by-step fashion in Algorithm 1. The calculations were verified on microbeads of different sizes and applied on cell images. The technique has a precision of about one

third wavelength. Compared to other microscopy techniques that were mentioned in Section 1 (e.g. PS-DIC and DPM), our technique provides a simple and cost-effective solution, it does not require additional microscope parts to be installed, and it reliably restores the phase information using just a single image.

References

[1] Alvarez, L., Lions, P.L., and Morel, J.M. Image selective smoothing and edge detection by nonlinear diffusion. II. *SIAM Journal on numerical analysis*, 29(3):845–866, 1992.

[2] Arnison, M.R., Cogswell, C.J., Smith, N.I., Fekete, P.W., and Larkin, K.G. Using the Hilbert transform for 3D visualization of differential interference contrast microscope images. *Journal of microscopy*, 199(1):79–84, 2000.

[3] Arnison, M.R., Larkin, K.G., Sheppard, C.J., Smith, N.I., and Cogswell, C.J. Linear phase imaging using differential interference contrast microscopy. *Journal of microscopy*, 214(1):7–12, 2004.

[4] Bhaduri, B., Edwards, C., Pham, H., Zhou, R., Nguyen, T.H., Goddard, L.L., and Popescu, G. Diffraction phase microscopy: principles and applications in materials and life sciences. *Advances in Optics and Photonics*, 6(1):57–119, 2014.

[5] Chan, T.F. and Vese, L. Active contours without edges. *IEEE transactions on Image processing*, 10(2):266–277, 2001.

[6] Cogswell, C.J., Smith, N.I., Larkin, K.G., and Hariharan, P. Quantitative DIC microscopy using a geometric phase shifter. In *BiOS'97, Part of Photonics West*, pages 72–81. International Society for Optics and Photonics, 1997.

[7] Cuche, E., Marquet, P., and Depeursinge, C. Simultaneous amplitude-contrast and quantitative phase-contrast microscopy by numerical reconstruction of Fresnel off-axis holograms. *Applied optics*, 38(34):6994–7001, 1999.

[8] Feineigle, P.A., Witkin, A.P., and Stonick, V.L. Processing of 3D DIC microscopy images for data visualization. In *Acoustics, Speech, and Signal Processing*, volume 4, pages 2160–2163. IEEE, 1996.

[9] Heise, B., Sonnleitner, A., and Klement, E.P. DIC image reconstruction on large cell scans. *Microscopy research and technique*, 66(6):312–320, 2005.

[10] Hooke, R. *Micrographia: or some physiological descriptions of minute bodies made by magnifying glasses, with observations and inquiries thereupon.* Courier Corporation, 2003.

[11] Kagalwala, F. and Kanade, T. Reconstructing specimens using DIC microscope images. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 33(5):728–737, 2003.

[12] Kim, T., Zhou, R., Goddard, L.L., and Popescu, G. Solving inverse scattering problems in biological samples by quantitative phase imaging. *Laser & Photonics Reviews*, 10(1):13–39, 2016.

[13] King, S.V., Libertun, A.R., Preza, C., and Cogswell, C.J. Calibration of a phase-shifting DIC microscope for quantitative phase imaging. In *Proc. SPIE*, volume 6443, page 64430M, 2007.

[14] Koos, K., Molnár, J., and Horvath, P. DIC Microscopy Image Reconstruction Using a Novel Variational Framework. In *Digital Image Computing: Techniques and Applications*, pages 1–7. IEEE, 2015.

[15] Koos, K., Molnár, J., Kelemen, L., Tamás, G., and Horvath, P. DIC image reconstruction using an energy minimization framework to visualize optical path length distribution. *Scientific Reports*, 6, 2016.

[16] Kou, S.S., Waller, L., Barbastathis, G., and Sheppard, C.J. Transport-of-intensity approach to differential interference contrast (TI-DIC) microscopy for quantitative phase imaging. *Optics letters*, 35(3):447–449, 2010.

[17] Li, K. and Kanade, T. Nonnegative mixed-norm preconditioning for microscopy image segmentation. In *International Conference on Information Processing in Medical Imaging*, pages 362–373. Springer, 2009.

[18] Mann, C., Yu, L., Lo, C.M., and Kim, M. High-resolution quantitative phase-contrast microscopy by digital holography. *Optics Express*, 13(22):8693–8698, 2005.

[19] Marquet, P., Rappaz, B., Magistretti, P.J., Cuche, E., Emery, Y., Colomb, T., and Depeursinge, C. Digital holographic microscopy: a noninvasive contrast imaging technique allowing quantitative visualization of living cells with subwavelength axial accuracy. *Optics letters*, 30(5):468–470, 2005.

[20] Mehta, S.B. and Oldenbourg, R. Image simulation for biological microscopy: microlith. *Biomedical optics express*, 5(6):1822–1838, 2014.

[21] Mehta, S.B. and Sheppard, C.J. Partially coherent image formation in differential interference contrast (DIC) microscope. *Optics express*, 16(24):19462–19479, 2008.

[22] Mehta, S.B. and Sheppard, C.J. Equivalent of the point spread function for partially coherent imaging. *Optica*, 2(8):736–739, 2015.

[23] Mumford, D. and Shah, J. Optimal approximations by piecewise smooth functions and associated variational problems. *Comm. Pure Appl. Math.*, 42(5):577–685, 1989.

[24] Murphy, D.B. and Davidson, M.W. *Differential interference contrast microscopy and modulation contrast microscopy. Fundamentals of Light Microscopy and Electronic Imaging, Second Edition p. 173-197.* 2001.

[25] Murphy, D.B. and Davidson, M.W. *Phase contrast microscopy and darkfield microscopy. Fundamentals of Light Microscopy and Electronic Imaging, Second Edition p. 115-133.* 2001.

[26] Osher, S.and Sethian, J.A. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of computational physics*, 79(1):12–49, 1988.

[27] Preza, C., King, S.V., and Cogswell, C.J. Algorithms for extracting true phase from rotationally-diverse and phase-shifted DIC images. In *Biomedical Optics 2006*. International Society for Optics and Photonics, 2006.

[28] Preza, C., Snyder, D.L., and Conchello, J.A. Theoretical development and experimental evaluation of imaging models for differential-interference-contrast microscopy. *JOSA A*, 16(9):2185–2199, 1999.

[29] Rudin, L.I., Osher, S., and Fatemi, E. Nonlinear total variation based noise removal algorithms. *Phys. D*, 60:259–268, November 1992.

[30] Schierbeek, A. *Measuring the invisible world: the life and works of Antoni van Leeuwenhoek.* Abelard-Schuman, 1959.

[31] Van Munster, E.B., Van Vliet, L.J., and Aten, J.A. Reconstruction of optical pathlength distributions from images obtained by a wide-field differential interference contrast microscope. *Journal of Microscopy*, 188(2):149–157, 1997.

[32] Vonesch, C., Aguet, F., Vonesch, J.L., and Unser, M. The colored revolution of bioimaging. *Signal processing magazine*, 23(3):20–31, 2006.

[33] Waller, L., Tian, L., and Barbastathis, G. Transport of intensity phase-amplitude imaging with higher order intensity derivatives. *Optics express*, 18(12):12552–12561, 2010.

[34] Waller, L., Tsang, M., Ponda, S., Yang, S.Y., and Barbastathis, G. Phase and amplitude imaging from noisy images by Kalman filtering. *Optics express*, 19(3):2805–2815, 2011.

[35] Yin, Z. and Kanade, T. Restoring DIC microscopy images from multiple shear directions. In *Biennial International Conference on Information Processing in Medical Imaging*, pages 384–397. Springer, 2011.

Acta Cybernetica **23** (2017) 645–658.

Balanced Active Learning Method for Image Classification

Dávid Papp[a] and Gábor Szűcs[a]

Abstract

The manual labeling of natural images is and has always been painstaking and slow process, especially when large data sets are involved. Nowadays, many studies focus on solving this problem, and most of them use active learning, which offers a solution for reducing the number of images that need to be labeled. Active learning procedures usually select a subset of the whole data by iteratively querying the unlabeled instances based on their predicted informativeness. One way of estimating the information content of an image is by using uncertainty sampling as a query strategy. This basic technique can significantly reduce the number of label needed; e.g. to set up a good model for classification. Our goal was to improve this method by balancing the distribution of the already labeled images. This modification is based on a novel metric that we present in this paper. We conducted experiments on two popular data sets to demonstrate the efficiency of our proposed balanced active learning (BAL) approach, and the results showed that it outperforms the basic uncertainty sampling.

Keywords: active learning, image classification, uncertainty sampling, balanced method

1 Introduction

Nowadays, for categorizing the huge amounts of visual contents both online and offline, image classification is an indispensable tool. The number of images available online is increasing with the rapidly growing Internet usage. Besides this numerous electronic devices are capable of taking a digital picture (e.g. cameras, telephones and so on), furthermore, smart devices are a click away from uploading and sharing these pictures. This leads to massive data warehouses that need to be structured, i.e. categorized. The classification of images requires labeled instances, but usually the contents are unlabeled, and labeling them is a costly manual process. Active learning [2] is a way of addressing this problem since it selects a subset of the data

[a]Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Magyar Tudósok krt. 2., H-1117, Budapest, Hungary.
E-mail: {pappd,szucs}@tmit.bme.hu

DOI: 10.14232/actacyb.23.2.2017.13

by iteratively querying the most informative image(s) from the unlabeled ones, and then it builds the classification model based on this subset instead of the whole data. In this way, the active learning algorithm seeks to label as few instances as possible (i.e., minimizing the labeling cost) while it attempts to retain the same level of accuracy that could be achievable by using the total dataset. The most important question is how to estimate the informativeness of unlabeled instances, since different query strategies may lead to a better or worse classification accuracy, compared to that using random sampling. There are many proposed strategies in the literature, e.g. uncertainty sampling [33], query-by-committee [31], expected model change [6], expected error reduction [20]. Uncertainty sampling is a widely used, and the simplest query strategy framework, which aims to query the instances with least certainty about their labels.

The main goal of our study was to improve the accuracy by enhancing it with a distribution analysis on the labeled dataset. We developed and implemented a solution to determine the distribution based on a novel penalty metric. We demonstrated the efficiency of the proposed approach on two large datasets; namely, the PASCAL VOC2010 [10] training and validation data, and the Caltech101 [17] image collection (see Section 5 for details and Figure 1 for sample images). Note that the queried images could be re-annotated by relying on human labor to deliver a more useful feedback on the effectiveness of the proposed approach in a production-like active learning environment. Of course, this approach would be resource-intensive, hence we decided to use a viable substitute to measure the benefits of the BAL algorithm (where we used the ground truth categorizations provided by the datasets).

2 Related Work

Now, we will briefly review the related studies on active learning. We will focus on computer vision problems (especially on images) [6, 13, 31, 33] that require substantial amounts of training data to perform accurate classification. Nowadays active learning is gaining increasing interest in the computer vision community. In the following, we review relevant work on active learning and image classification. Some of these studies are only theoretical ones [7] without any experimental part or results. The authors of [32] used simple margin selection method for SVM for the active selection of object windows in images. The system autonomously refines its models by actively requesting crowd-sourced annotations on images crawled from the Web. The authors of [24] suggested combining spectral and spatial information directly in the iterative process of sample selection, where the criterion involves the concept of spatial entropy.

As we already mentioned in the Introduction, uncertainty sampling is a frequently adopted strategy in active learning (because of its simplicity), which builds upon the presence of uncertainty in classification. Minakawa et al. [23] apply this uncertainty sampling method to image sequence recognition in an active learning scenario; a margin sampling criterion and entropy criterion were used in the condition part of this method. Many other works on uncertainty sampling methods are

Figure 1: Samples taken from the PASCAL 1 test set; here each column corresponds to a different category; namely, airplane, boat, cat, motorbike and sheep, respectively

based on the entropy notion. In a recent paper [33], the authors evaluate the uncertainty via random walks on a graph, and Shannon Entropy is used to measure the uncertainty of random variables in this random process. We compared our method with the most frequently used uncertainty sampling technique (based on entropy criterion) in this paper. Another solution called the DBALStream method [14] is based on uncertainty sampling as well for active learning over evolving stream data. This approach has to decide whether to label an instance or not, but the method works in only the kind of scenario where each instance in the stream is processed as soon as it arrives. In the uncertainty sampling topic, one of the most recent paper [28] attempts to distinguish between the two types of uncertainties (conflicting-evidence vs. insufficient-evidence), but it does not provide another alternative approach for improving uncertainty sampling. One possible drawback of the uncertainty sampling algorithms is that they ignore the output distributions for the remaining class labels [12]; however our solution attempts to overcome this problem by a proposed balancing extension.

3 Image Classification

In the case of active learning, we iteratively query new images to expand the training set, and build a new classification model every time the training set changes. The decision of which image is next in line to be queried depends on the current model,

therefore the model creation process is an important part of the classification and the quality of the classification is an important part of the active learning. Following the general trend, we applied the BoW (Bag-of-Words) model [11, 16, 18] for the mathematical representation of the images and SVMs (Support Vector Machines) [1, 8, 16] for the classification.

The key idea behind the BoW model is to represent an image (based on its visual content) with so-called visual code words while ignoring their spatial distribution. This technique consists of three steps, these being (i) feature detection, (ii) feature description and (iii) image description as usual phases in computer vision. For feature detection we utilized the Harris-Laplace corner detector [4, 22], and SIFT (Scale Invariant Feature Transform) [19] to describe them. We should add that we used the default parameterization of SIFT proposed by Lowe; hence we got descriptor vectors with 128 dimensions. To define the visual code words from the descriptor vectors, we used the GMM (Gaussian Mixture Model) [25, 30], which is a parametric probability density function represented as a weighted sum of (in our case 256) Gaussian component densities. As can be seen below,

$$p(X \mid \lambda) = \sum_{j=1}^{K} \omega_j g(X \mid \mu_j o_j), \tag{1}$$

where μ_j and o_j denote the expected value and the variance of the j^{th} Gaussian component, respectively, and here $K = 256$. We calculated the λ parameter with the ML (Maximum Likelihood) estimation using the iterative EM (Expectation Maximization) algorithm [9, 30]. We performed K-means clustering [21] over all the descriptors with 256 clusters to get the initial parameter model for the EM. Next, we had to create a descriptor that specifies the distribution of the visual code words in an image called the high-level descriptor. To represent an image with a high-level descriptor, the GMM-based Fisher vector was calculated, as can be seen in Equation 2. These vectors were the final representations (image descriptor) of the images.

$$F = \bigtriangledown_\lambda \log p(X \mid \lambda). \tag{2}$$

For the classification subtask, we used a variation of SVM called the C-SVC (C-support vector classification) [1, 8] with a RBF (Radial Basis Function) kernel. Furthermore we applied the one-against-all approach to extend the SVM to the multi-class classification case.

4 Proposed Approach

Uncertainty sampling is the most common active learning query strategy framework, which queries the instances with least certainty about their labels. There are numerous ways to measure the amount of uncertainty, the easiest one being to query the instance whose prediction is the least confident. The problem with this approach is that it just processes information about the most probable class,

and throws away information about the remaining label distribution. On the other hand, a different uncertainty sampling variant called margin-sampling takes the first and second most probable class labels, but this still ignores many information. More general and probably the most popular uncertainty sampling strategies use entropy as an uncertainty measure, and query the instance with the maximum one.

The basis of our approach is this variation of uncertainty sampling. As we mentioned previously, we used SVM for classification. According to the literature, querying the closest instance to the linear decision boundary is analogous to uncertainty sampling with a probabilistic binary linear classifier [3, 13, 27]. SVM is basically not a probabilistic classifier so we applied a variation of Platt's [15] approach as a probability estimator to get the probability values (confidence values) of the possible labels. This approach is included in LIBSVM [5, 29], and we calculated the entropy based on it by using

$$H_j = -\sum_{i=1}^{m} P(l_i \mid t_j) \times \log P(l_i \mid t_j), \tag{3}$$

where H_j denotes the entropy of the j^{th} unlabeled image, m is the number of categories and $P(l_i \mid t_j)$ denotes the confidence value that l_i is the label of image t_j. Furthermore, we defined a novel so-called penalty metric as

$$Penalty_j = CTR_{j^*} \times \frac{1}{m}, \tag{4}$$

where j^* denotes the estimated category of the j^{th} unlabeled image, and CTR_{j^*} denotes a general counter whose value increases by 1 with each iteration of queries when the received category is other then j^*. Thus each unlabeled image possess a penalty value depending on its predicted category. Here, we normalized the CTR values by the number of classes (m), hence the penalty metric also depends on the number of classes. When the learning system queries a particular category (i.e. an image from that category), the penalty value of each unlabeled image we predict to be in that category is reset to zero. Within each cycle, we merge the actual penalty values with the ones coming from uncertainty sampling, and these results will give the final decision scores. The fusion was made based on a β weighting coefficient in the following formula:

$$HP_j = (1 - \beta) \times H_j + \beta \times Penalty_j, \tag{5}$$

where HP_j denotes the informativeness of the j^{th} unlabeled image. The higher the HP score of an image is, the more likely it will query its label. Therefore, we sorted the images by their HP values in descending order and chose the first (few) candidate images for labeling. In order to keep the same value of β for different data sets or tasks (i.e. the different number of classes), we decided to use the kind of normalization we described in Equation 4. The advantage of our proposed approach is that it strikes a balance among the classes of labeled instances, especially when the number of classes is very high, perhaps a hundred or more (see Section 5). In the following, we present our balanced active learning (BAL) algorithm in a pseudo

code (see Algorithm 1); note that in the third line, the *"terminating condition is false"* statement should be replaced with the desired one (which in our case was $size(I_L) \neq size(I_U)$).

Algorithm 1 Balanced Active Learning Algorithm (BAL)

input: Unlabeled image set I_U
initialize: Create the initial labeled image set I_L. $\forall C \in I_U$: select an image
while (terminating condition is false) **do**
  Train SVM model M based on the actual I_L
  for $\forall$ image $j \in I_U$ **do**
    Calculate class membership probabilities ($P(C \mid j)$) using M
    Calculate the entropy (H_j) using Equation 3
    Calculate the penalty metric ($Penalty_j$) using Equation 4
    Merge the entropy with the penalty metric (HP_j) using Equation 5
  end for
  sort: $\forall$ image $j \in I_U$ by HP_j in descending order
  query: image j_q from top of the list: $j_q = query(argmax_{j \in I_U} HP_j)$
  insert: j_q and its real label l into I_L: $I_L = I_L \cup \{j_q, l\}$
  remove: j_q and its real label l from I_U: $I_U = I_U \setminus \{j_q, l\}$
  reset: penalty metrics based on the learned real label: $\forall j \in C_l : Penalty_j = 0$
end while

5 Experimental Results

5.1 Experimental Environment

Next, we will present the experimental results of our proposed balanced active learning query strategy. We evaluated three strategies for comparison. These were

- Random sampling
- Uncertainty sampling using entropy as measure
- The proposed balanced active learning (BAL) approach. That is, uncertainty sampling using an entropy metric merged with a novel penalty metric calculated from the distribution of recently queried class labels (see Algorithm 1)

We used two large and popular image classification data sets in our experiments, namely the PASCAL VOC2010 [10] training and validation datasets, and the Caltech101 [17] image collection. The PASCAL VOC2010 consists of 20 classes and it has 10,103 images in total. We randomly selected 50 images from each category to form a subset of this data set. The Caltech101 collection contains 8677 images taken from 101 different (real) categories and an additional noise category. We

discarded the noise category to simplify the data set and randomly chose 30 images from each class to create our second subset. Afterwards, we randomly selected 10 categories from the PASCAL and 20 categories from the Caltech101 subsets to form two additional subsets. The following table presents the details concerning our chosen data sets; and in Figure 1 we can see some sample images from the PASCAL 1 test set.

Table 1: Details of the randomly selected subsets and imbalanced data sets

	Number of classes	Number of runs	Number of images
CALTECH101 1	20	10	600
CALTECH101 2	101	10	3030
CALTECH101 3	20	1	1961
CALTECH101 4	101	1	8677
PASCAL 1	10	10	500
PASCAL 2	20	10	1000

The distribution of images among the classes were even in the created subsets, but our approach is also capable of handling imbalanced data sets. Therefore we included two more test sets (CALTECH101 3 and CALTECH101 4) those consisted of the same categories as CALTECH101 1 and CALTECH101 2, but the number of images were not reduced (i.e. CALTECH101 4 was the whole Caltech101 dataset).

During the experiments, each queried image and its label was moved to the training set from the test set. For the initial training set, we randomly selected an image from each category. In the last iteration, the size of the training set and test set were the same; e.g. in the case of PASCAL 1 data set, the learning system stopped querying the unlabeled instances when the number of training images reached 250. As can be seen in the third column of Table 1, we performed the experiments several times with the same data set, and we averaged out the results of the separate runs.

5.2 Results

Now we will turn to the results. Within each cycle, we evaluated the accuracy and MAP (Mean Average Precision) [26] metrics on the actual test set. After a series of tests, we chose β to be 0.1 (see Equation 5); because this gave a sufficient weight for the penalty metric to positively influence the entropy. It should be added that the proposed approach is highly sensitive to the choice of β (see Figure 5 for details). The overall results of the generated subsets of the Caltech101 collection and the PASCAL data set can be seen in figures 2 and 3, respectively; and the results of the unbalanced test sets can be seen in Figure 4. As we mentioned previously, each data set was tested several times to be able to take the average of the separate runs and thus give a more trustworthy result. This was important because we commenced each of the tests with different initial training sets, since the first images were

randomly selected. In the graphs we can see the averages, and for our proposed approach, we also included error bars to show the minimum and maximum values at each sampling point. This presents the variance of the BAL algorithm, as can be seen in the following figures.

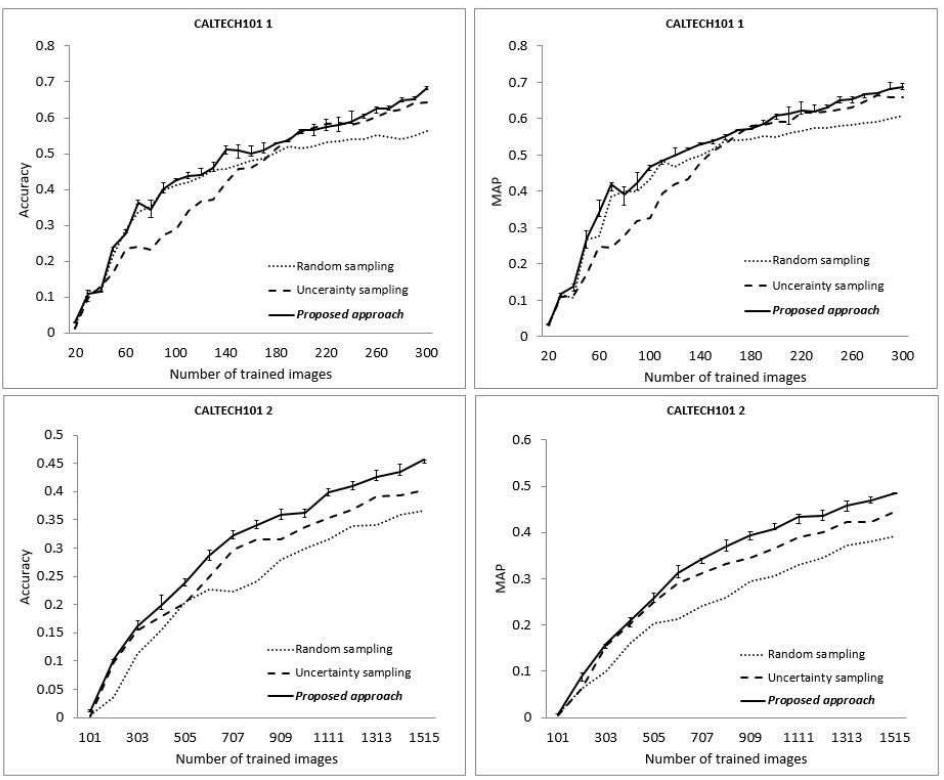


Figure 2: Evaluation of the results obtained on the CALTECH101 1 and CALTECH101 2 datasets. The accuracy and MAP metrics are on the y-axis, while the number of trained images is on the x-axis. The proposed approach, uncertainty sampling and random sampling are represented by solid, dashed and dotted lines, respectively.

As the results of the CALTECH101 1 test set in Figure 2 show, the distribution analysis has a beneficial effect. The basic uncertainty sampling with entropy (represented by a dashed line) yielded worse results at the first 150 iterations. This was because it queried many instances from the same category, but this could not happen with the proposed approach, since it balances the label distribution. At the last sampling point, our approach gave better results than the other competitor methods. In Figure 2, the results of the CALTECH101 2 test set are also presented. As can be seen, the balanced active learning algorithm consistently outperforms the

other two approaches. Based on this, we may conclude that the balancing technique is more advantageous when the number of classes is relatively high.

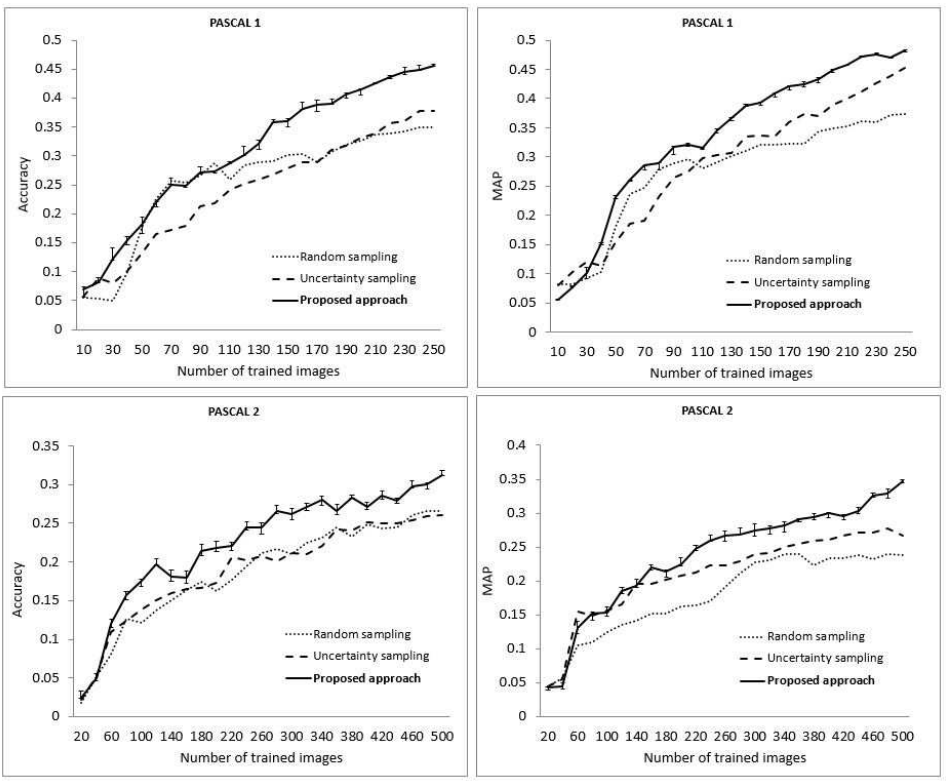


Figure 3: Evaluation of the results obtained on the PASCAL 1 and PASCAL 2 datasets. The accuracy and MAP metrics are on the y-axis, while the number of trained images is on the x-axis. The proposed approach, uncertainty sampling and random sampling are represented by solid, dashed and dotted lines, respectively.

The results of the PASCAL 1 and PASCAL 2 test sets are shown in Figure 3; in the graphs we can discern some similarities in the pattern of the curves to those in Figure 2. For example, the proposed approach gave the highest metrics among the three methods for almost each sampling point for both test sets; although if we look at the accuracy values of PASCAL 2, we see that the difference in the results is higher and less volatile between our BAL approach and the other two methods. This observation seems to support our hypothesis about the existence of a proportional relationship between the performance of BAL algorithm and the number of categories.

The results of CALTECH101 3 and CALTECH101 4 can be seen in Figure 4. In this case, the accuracy and MAP values were higher for the last sampling point,

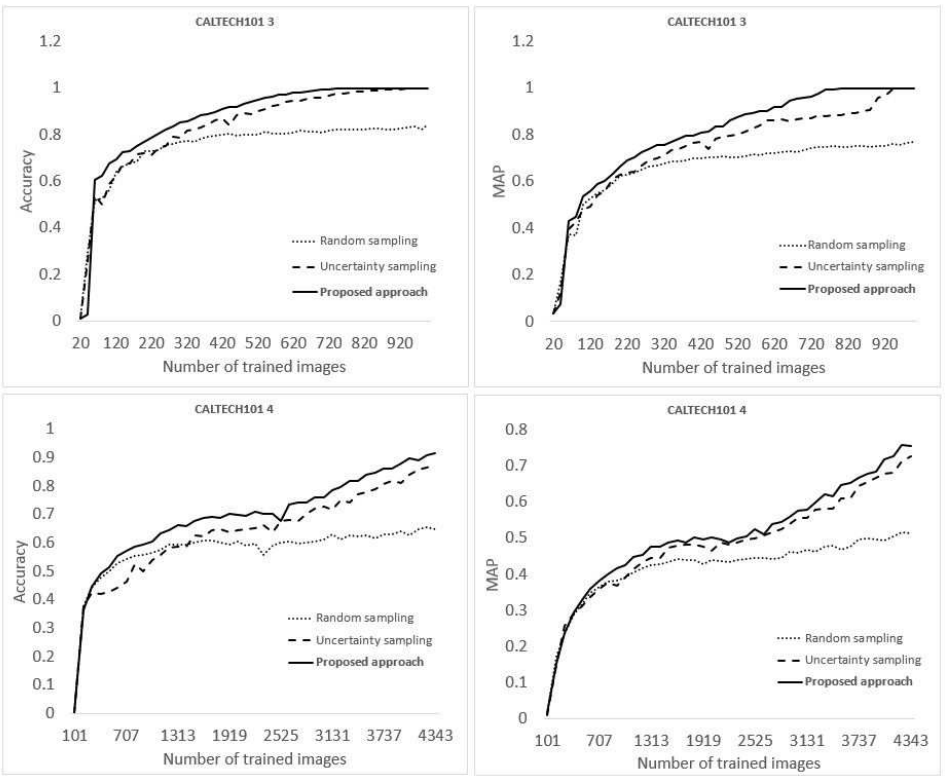


Figure 4: Evaluation of the results got on the CALTECH101 3 and CALTECH101 4 datasets. The accuracy and MAP metrics are on the y-axis, while the number of trained images is on the x-axis. The proposed approach, uncertainty sampling and random sampling are represented by solid, dashed and dotted lines, respectively.

because the learning system had more training images. Although this difference decreased, the proposed approach outperforms the other two competitor methods tested here almost at every sampling point (except the first few at CALTECH101 3). However, the BAL algorithm attains an accuracy of 1.0 and MAP of 1.0 faster (i.e. with fewer training images) than the general uncertainty sampling; therefore our proposed approach reduces the number of required training instances to achieve the maximum possible classification accuracy, which is the main goal of our study (and using active learning in general).

We summarized the result obtained of all our experiments in Table 2; the accuracy and MAP values shown in the table were measured for the last iteration. As can be seen, the proposed approach outperformed the other query strategies in every case here.

In the last experiment, we give a brief explanation of our choice of β (see

Table 2: Summary of the accuracy and MAP values got on the four test sets

	Rand. sampling		Unc. sampling		**BAL approach**	
	Acc.	MAP	Acc.	MAP	Acc.	MAP
CALTECH101 1	0.562	0.609	0.642	0.660	**0.683**	**0.688**
CALTECH101 2	0.366	0.390	0.401	0.445	**0.456**	**0.484**
CALTECH101 3	0.842	0.770	0.998	0.997	**1.000**	**1.000**
CALTECH101 4	0.648	0.514	0.876	0.725	**0.916**	**0.755**
PASCAL 1	0.350	0.374	0.378	0.453	**0.455**	**0.482**
PASCAL 2	0.266	0.238	0.260	0.266	**0.312**	**0.347**

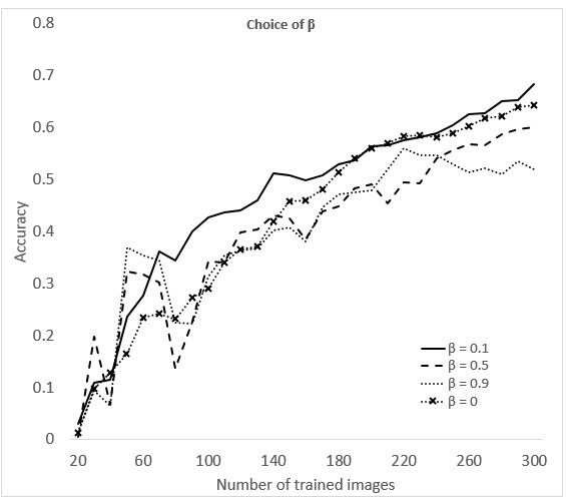


Figure 5: Sensitivity for the choice of β on the CALTECH101 1 dataset. Accuracy metric is on the y-axis, while the number of trained images is on the x-axis. The cases $\beta = 0.1$, $\beta = 0.5$ and $\beta = 0.9$ are represented by solid, dashed and dotted lines, respectively.

Figure 5). We used the CALTECH101 1 data set for this test, and we evaluated the results for the following β values: 0.1, 0.5, 0.9 and 0. Here, $\beta = 0$ is a special case, since it returns the general uncertainty sampling with an entropy measure (i.e. the dashed line in the upper left sub-figure of Figure 2). This figure tells us that the proposed approach is highly sensitive to the choice of β, because with the wrong values it is possible to achieve a worse performance than that using the simple entropy measure.

6 Conclusions

In this paper we presented a new query strategy for active learning, which is an improvement of the basic uncertainty sampling technique. The proposed approach combines the entropy measure with a novel penalty metric to balance the class distribution of labeled instances. This modification solves a possible deficiency of the uncertainty sampling, the skipping of certain categories. We employed two large data sets (PASCAL VOC2010 and Caltech101) to demonstrate the efficiency of our approach, and we evaluated the accuracy and MAP metrics. Our experiments demonstrated that the proposed balanced active learning (BAL) approach generally outperforms the random sampling and the basic uncertainty sampling methods.

References

[1] Boser, B., Guyon, I. and Vapnik, V. A Training Algorithm for Optimal Margin Classifier. *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pp. 144-152, 1992.

[2] Burr Settles *Active learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning, Vol. 6, No. 1, pp. 1-114, 2012.

[3] C. Campbell, N. Cristianini and A. Smola Query learning with large margin classifiers. *Proceedings of the 17th International Conference on Machine Learning*, pp. 111-118, 2000.

[4] C. Harris, M. Stephens A combined corner and edge detector. *Proceedings of the Alvey Vision Conference*, pp. 23.1-23.6, 1988.

[5] C.-C. Chang, C.-J. Lin LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, Vol. 2, No. 3, pp. 27.1-27.27, 2011.

[6] Cai, W., Zhang, Y. and Zhou, J. Maximizing expected model change for active learning in regression. *IEEE 13th International Conference on Data Mining*, pp. 51-60, 2013.

[7] Chaudhuri, K., Kakade, S. M., Netrapalli, P. and Sanghavi, S. Convergence rates of active learning for maximum likelihood estimation. *Advances in Neural Information Processing Systems 28*, pp. 1090-1098, 2015.

[8] Cortes, C., Vapnik, V. Support-vector networks. *Machine Learning*, Vol. 20, No. 3, pp. 273-297, 1995.

[9] Dempster, A., Laird, N. and Rubin, D. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, Vol. 39, No. 1, pp. 1-38, 1977.

[10] Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J. and Zisserman, A. The PASCAL Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, Vol. 88, No. 2, pp. 303-338, 2010.

[11] Fei-Fei, L., Fergus, R. and Torralba, A. Recognizing and Learning Object Categories. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[12] Fu, Y., Zhu, X. and Li, B. A survey on instance selection for active learning. *Knowledge and Information Systems*, Vol. 35, No. 2, pp. 249-283, 2013.

[13] G. Schohn, D. Cohn Less is more: Active learning with support vector machines. *Proceedings of the 17th International Conference on Machine Learning*, pp. 839-836, 2000.

[14] Ienco, D., Zliobaite, I. annd Pfahringer, B. High density-focused uncertainty sampling for active learning over evolving stream data. *Proceedings of the 3rd International Workshop on Big Data*, pp. 133-148, 2014.

[15] J. Platt Probabilistic outputs for support vector machines and comparison to regularize likelihood methods. *Advances in Large Margin Classifiers*, pp. 61-74, 2000.

[16] K. Chatfield, V. Lempitsky, A. Vedaldi and A. Zisserman The devil is in the details: an evaluation of recent feature encoding methods. *Proceedings of the 22nd British Machine Vision Conference*, pp. 76.1-76.12, 2011.

[17] L. Fei-Fei, R. Fergus and P. Perona Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Workshop on Generative-Model Based Vision, 2004.

[18] Lazebnik, S., Schmid, C. and Ponce, J. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Vol. 2, pp. 2169-2178, 2006.

[19] Lowe, D. G. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, Vol. 60, No. 2, pp. 91-110, 2004.

[20] Mac Aodha, O., Campbell, N., Kautz, J. and Brostow, G. Hierarchical subquery evaluation for active learning on a graph. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 564-571, 2014.

[21] MacQueen, J. Some methods for classification and analysis of multivariate observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1, pp. 281-297, 1967.

[22] Mikolajczyk, K., Schmid, C. Scale & affine invariant interest point detectors. *International Journal of Computer Vision*, Vol. 60, No. 1, pp. 63-86, 2004.

[23] Minakawa, M., Raytchev, B., Tamaki, T. and Kaneda, K. Image sequence recognition with active learning using uncertainty sampling. *Proceedings of the International Joint Conference on Neural Networks*, pp. 1-6, 2013.

[24] Pasolli, E., Melgani, F., Tuia, D., Pacifici, F. and Emery, W. J. SVM active learning approach for image classification using spatial information. *Proceedings of the IEEE Transactions on Geoscience and Remote Sensing*, Vol. 52, No. 4, pp. 2217-2233, 2014.

[25] Reynolds D. A. *Gaussian Mixture Models.* Encyclopedia of Biometric Recognition, pp. 659-663, 2009.

[26] Robertson, S. A new interpretation of average precision. *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 689-690, 2008.

[27] S. Tong, D. Koller Support vector machine active learning with applications to text classification. *Proceedings of the 17th International Conference on Machine Learning*, pp. 9991006, 2000.

[28] Sharma, M., Bilgic, M. *Evidence-based uncertainty sampling for active learning.* Data Mining and Knowledge Discovery, pp. 1-39, 2016.

[29] T.-K. Huang, R. C. Weng and C.-J. Lin Generalized Bradley-Terry models and multi-class probability estimates. *Journal of Machine Learning Research*, Vol. 7, pp. 85-115, 2006.

[30] Tomasi C. Estimating Gaussian mixture densities with EM - A Tutorial. Technical report, Duke University, 2004.

[31] Tsai, Y. L., Tsai, R. T. H., Chueh, C. H. and Chang, S. C. Cross-Domain Opinion Word Identification with Query-By-Committee Active Learning. *Proceedings of the 19th International Conference on Technologies and Applications of Artificial Intelligence*, pp. 334-343, 2014.

[32] Vijayanarasimhan, S., Grauman, K. Large-scale live active learning: Training object detectors with crawled data and crowds. *International Journal of Computer Vision*, Vol. 108, No. 1-2, pp. 97-114, 2014.

[33] Yang, Y., Ma, Z., Nie, F., Chang, X. and Hauptmann, A. G. Multi-class active learning by uncertainty sampling with diversity maximization. *International Journal of Computer Vision*, Vol. 113, No. 2, pp. 113-127, 2015.

Acta Cybernetica **23** (2017) 659–686.

Unit Testing in C++ with Compiler Instrumentation and Friends

Gábor Márton[a] and Zoltán Porkoláb[b]

Abstract

In C++, test code is often interwoven with the unit we want to test. During the test development process we often have to modify the public interface of a class to replace existing dependencies; e.g. a supplementary setter or constructor function is added for dependency injection. In many cases, extra template parameters are used for the same purpose. All existing solutions have serious detrimental effects on the code structure and sometimes on the run-time performance as well. In this paper, we overview existing dependency replacement techniques of C++ and we evaluate their advantages and disadvantages. We introduce our non-intrusive, compiler instrumentation based testing approach that does not have such disadvantages. All non-intrusive testing methods (including our new method) require access to an object's internal state in order to setup a test. Thus, to complement our new solution, we also present different approaches to conveniently access private members in C++. To evaluate these techniques, we created a proof-of-concept implementation which is publicly available for further testing.

Keywords: C++, unit test, instrumentation, friend, access control

1 Introduction

Testing is essential in modern software development [1, 13, 5, 18] to improve the quality of a system and reduce the cost of maintenance. There are different layers of testing from unit tests to stability, functional and integration tests. In this paper we focus on unit testing, which is the most language-specific method. However, some of the findings we discuss might be extended to different/higher level tests as well.

During a unit test we check the behaviour of the unit under test. If we do functional programming and work with pure functions alone (where all functions are free from side-effects) then testing is easy because we just provide a specific input and assert for the desired output. However, in the object-oriented paradigm,

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, E-mail: martongabesz@gmail.com

[b]E-mail: gsd@elte.hu

DOI: 10.14232/actacyb.23.2.2017.14

we have objects in some kind of state. This means the object is dependent on other objects that represent the internal state. Testing our object using these dependencies may be problematic; e.g. the dependency may represent a database or a network connection, whose behaviour can be hard or expensive to simulate. In order to create independent, resilient and efficient tests [45] in most cases we need to substitute some (or even all) of the dependencies with test doubles. We refer to this substitution of dependencies as *dependency replacement*.

In object-oriented programming languages, the dependency replacement often requires the modification of the original public interface of the unit under test. For instance, new setter or constructor functions have to be added to a class, otherwise dependency replacement would not work. Nevertheless, there are cases where these new functions are not intended to be used in production code. We refer to all those testing approaches which require source code modification as *intrusive* testing. Moreover, in C++, source code modification for testing could result in performance degradation, e.g. introducing a new runtime interface and virtual functions just because of testing might worsen the performance of the production code. Also, in legacy code bases often there are no unit tests. Refactoring such legacy code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, though all of the existing non-intrusive testing methods for C++ have some drawbacks.

In this paper, we investigate a new, non-intrusive, compiler instrumentation based testing approach that does not have these disadvantages. All non-intrusive testing methods (including our new method) often require access to an object's internal members in order to set up a test. Thus, to complement our new method, we present different approaches to conveniently access private members in C++.

This paper is organized as follows. In Section 2, we describe principles of dependency replacement in object-oriented programming and we discuss the existing techniques of dependency replacement in C++. We show how we can replace dependent C++ functions with compiler instrumentation in Section 3. Then the access of private members is discussed in Section 4. Here, we inspect how we can access members with our alternative approaches without intrusively changing the unit we wish to test. We present how we could enhance the use of friends with our extension idea to friends. After, we overview related work in Section 5. In Section 6, we outline future work and possible directions for better testing experience in C++. Our paper concludes in Section 7.

2 Dependency Replacement in C++

Figure 1 shows a typical object under test, its dependencies and their possible replacements. If A and B are objects and "A depends on B", then we say that A is a *dependant* of B and B is a *dependency* of A. As for dependency replacement the dependant object is referred as the *system under test* (SUT). Sometimes we refer to that as the *unit* under test. In this study, we use the following definitions

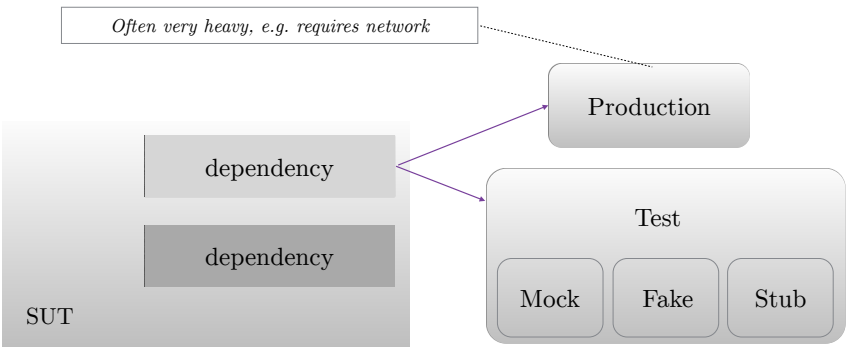


Figure 1: Dependency Replacement

for test doubles: *Fake* classes provide empty definitions of functions in a way that the unit tests can pass. Fakes are the simplest doubles to cut down dependencies. *Stub* classes implement additionally some very basic behaviour, therefore they may be more complex than fakes. We can set up a stub to return with a specific value. *Mock* classes are used to formulate expectations, such as how many times a member function is called with a certain value.

There are several design patterns for dependency replacement like the Factory Method and Abstract Factory [7], the Service Locator [44, 6] and the Dependency Injection (DI) [37, 38, 43].

It is important to emphasize that dependency injection is different from the abstract concept of dependency replacement. Dependency injection (DI) is one realization – amongst many others – of dependency replacement. DI is used mostly in object-oriented languages with runtime reflection, like Java and C#. All of these patterns can be used in the popular, managed languages and in C++ as well. Java and C# provides well documented DI frameworks (like the *Unity Container* in C# [29], and the Spring framework in Java [19]), therefore DI is the widespread method for performing dependency replacement in these managed languages. However, there is no generally accepted DI framework for C++.

Objects in the context of OOP are represented by `class`es in C++. Since C++ is not a strict object oriented language, we must investigate other language constructs like free functions and function templates from the viewpoint of dependency replacement. Generally speaking, a dependant C++ entity (class, function, class template or function template) can have different kinds of dependencies. For instance, it may have a dependency on

- a global object (e.g. via a singleton).
- a global function (via a function call),
- an object via a pointer or reference,
- a type (e.g. via a type template parameter, or the type of a member),

2.1 C++ Seams

A *seam* is an abstract concept introduced by Feathers [4] as an instrument via we can alter behaviour without changing the original unit. Dependency replacement is done via seams in C++. Actually, there are four different kinds of seams in C++ [35, 28]:

1. *Link seam*: Change the definition of a function via some linker specific setup.
2. *Preprocessor seam*: With the help of the preprocessor, redefine function names to use an alternative implementation.
3. *Object seam*: Based on inheritance to inject a subclass with an alternative implementation.
4. *Compile seam*: Inject dependencies at compile-time through template parameters.

The *enabling point* of a seam is the place where we can make the decision to use one behaviour or another. Different seams have different enabling points. For example, replacing the constructor argument for the implementation of an interface with a mock implementation when a unit test is set up is an object seam with the constructor as an enabling point.

2.1.1 Link Seams

We can use a link seam e.g. to replace the implementation of a free function or a member function. For instance:

```
// A.hpp
void foo();
// A.cpp
void foo() { ... };
// MockA.cpp
void foo() { ... };
// B.cpp
#include "A.hpp"
void bar() { foo(); ... }
```

On the one hand, when we need to test the `bar()` function then we should link the test executable to the `MockA.o` object file. On the other hand, we should link the production code with `A.o`. Link-time dependency replacement is not possible if the dependency is defined in a static library or in the same translation unit where the SUT is defined. It is also not feasible to use link seams if the dependency is implemented as an inline function [35]. This makes the use of this seam cumbersome or practically impossible when the dependant unit is a template or when the dependency is a template. The enabling point for a link seam is always outside of the program text. This makes the use of link seams quite difficult to identify. On top of all, link-time substitution requires strong support from the build system we are using. Thus, we might have to specialize the building of the tests for each and every unit. This does not scale well and can be really demanding regarding to maintenance.

2.1.2 Preprocessor Seams

Preprocessor seams can be applied to replace the invocation of a global function to an invocation of a test double [27]. Let us consider the following code snippet:

```
void *my_malloc(size_t size) {
  //...
  return malloc(size);
}

void my_free(void *p) {
  //...
  return free(p);
}

#define free my_free
#define malloc my_malloc

void unitUnderTest() {
  int *array = (int *)malloc(4 * sizeof(int));
  // do something with array
  free(array);
}
```

We can replace the standard `malloc()` and `free()` functions with our own implementation. One example usage may be to collect statistics or do sanity checks in `my_malloc` and `my_free` functions. These seams can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

2.1.3 Object Seams

Object seams are realized by introducing a runtime interface. For example, let us consider the following C++ class (note, using a `const` qualifier on the `process()` member function and a RAII lock guard instead of explicit locking would make the code safer, but it would also make our message less visible):

```
class Entity {
public:
  int process(int i) {
    if(m.try_lock()) {
      auto result = std::accumulate(v.begin(),v.end(),i);
      m.unlock();
      return result;
    }
    else { return -1; }
  }
  void add(int i) {
    m.lock();
    v.push_back(i);
    m.unlock();
  }
private:
  mutable std::mutex m;
  std::vector<int> v;
};
```

We would like to test the `Entity::process()` function for both possible return values of `try_lock`. Our objective is to have a test like this:

```
void test() {
  Entity e;
  set_try_lock_fails(e);
  ASSERT_EQUAL(e.process(1), -1);
  set_try_lock_succeeds(e);
  ASSERT_EQUAL(e.process(1), 1);
}
```

We introduced an interface and we can use it to change the behaviour of the mutex object in runtime. For this, the production specific and test specific implementations of the interface need to be provided:

```
struct Mutex {
  virtual void lock() = 0;
  virtual void unlock() = 0;
  virtual bool try_lock() = 0;
  virtual ~Mutex() {}
};

struct RealMutex : Mutex { //used in production code
  void lock() override { m.lock(); }
  void unlock() override { m.unlock(); }
  bool try_lock() override { return m.try_lock(); }
private:
  std::mutex m;
};

struct StubMutex : Mutex { //used in test code
  // defintion of lock() and unlock() as before
  bool try_lock_result = false;
  bool try_lock() override {
    return try_lock_result;
  }
};
```

Now our class should use the interface:

```
class Entity {
public:
  Entity(Mutex& m) : m(m) {}
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex& m;
  //...
};
```

We can see that the enabling point of this seam is the newly added constructor. The production and the test code might look like this:

```
void productionClient() {
  RealMutex m;
  Entity e(m);
  // some usage of e
}
void testClient() {
  // test setup
  StubMutex m;
  Entity e(m);
  // assertions ...
}
```

There is a severe problem with object seams that is illustrated via this specific example: the mutex object which was exclusively owned by the `Entity` now is moved outside. There is nothing to prevent any caller from reusing (misusing) this mutex. Regarding encapsulation this is fatal. Also, it is not clear who should create/destroy this object and when. The same problems arise if we replace the `Mutex&` with a raw pointer or a smart pointer. Though passing a `unique_ptr` in constructor and getting a reference to it via a getter might be an option, but then we would need to have a getter function for `m` (to set up the test). Generally speaking, the following problems may arise when we replace dependency objects:

- Either we deprive the unit under test from the ownership of the dependency or we use a superfluous getter function.
- We add an otherwise unnecessary constructor or setter function.
- We introduce superfluous pointer semantics via a reference or smart pointer, which is harmful to cache locality, hence it reduces overall performance [41].
- We have to introduce an interface just for testing. This interface has virtual functions. Calling them requires extra pointer indirections and this might result in cache misses and it loses the possibility of inlining, thus it harms the overall performance [3]. Adding an extra interface makes the program more complex, hence the program is harder to understand. Note that in some cases it might be possible to get rid of the additional explicit interface definition with type erasure [31], but the virtual function calls cannot be avoided even in this case.

2.1.4 Compile Seams

Our `Entity` and mutex example would be more natural if we make `Entity` a template and we use the `Mutex` type as a template parameter:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex m;
  //...
};
```

However, because of testing we need to access the mutex outside of the `Entity` class. Therefore, one simple approach is to define a getter function:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  Mutex& getMutex() { return m; } // Use only from tests
  //...
private:
  Mutex m;
  //...
};
```

The enabling point of this seam is the template parameter itself. Client code may use our `Entity` with the appropriate type parameter:

```
void productionClient() {
  Entity<std::mutex> e;
  // some usage of e
}
void testClient() {
  struct StubMutex {
    //...
    bool try_lock_result = false;
    bool try_lock() {
      return try_lock_result;
    }
  };
  Entity<StubMutex> e;
  auto& m = e.getMutex();
  m.try_lock_result = false;
  ASSERT_EQUAL(e.process(1), -1);
  m.try_lock_result = true;
  ASSERT_EQUAL(e.process(1), 1);
}
```

We do not need to add an additional runtime interface this time, so the test client can use a `StubMutex` which does not have any virtual functions. Of course, the implicit compile-time interface [25, item 41] of `std::mutex` and `StubMutex` must match.

With this approach, we introduced a template parameter just because of testing. The original `Entity` however was perfectly natural to be a simple class, now it became a class template. Also, we added an extra getter function to be able to drive the dependency externally from our class. Needless to say, we increased code complexity and compilation time [2]. There are some methods with which we could decrease compile time, but they would further complicate the source code (use of pimpl [26, item 22]) or the build system setup (using extern templates [12, 14.7.2]).

All four seams have some disadvantages that prevents us from using them or make us reluctant to use them. Link seams do not work with inline functions and require patching the build system. Preprocessor seams are problematic with classes and namespaces. Object seams and compile seams are intrusive and often demand that we widen the public interface. Also, object seams might introduce additional performance penalty in the production code. Therefore, we seek for a new seam which does not have the above-mentioned disadvantages. Our contribution is to show that such a seam can be implemented and applied in software testing.

3 Compiler Instrumentation for Testing

In software development, it is normal that the compiler generates different code for verification purposes. In most integrated development environments (IDEs), a project has two build configurations: namely debug and release. The debug profile usually does not define the `NDEBUG` macro; this results in a runtime binary which does check all the assertions passed to the `assert` macro [11, 7.2 Diagnostics]. Also, in the debug profile, the debug symbols are in general attached to the binary and

the compiler optimizations are turned off. There is a new proposal for C++ that proposes a minimal system for expressing interface requirements as contracts [34]. Contracts are requirements that an operation places on its arguments for successful completion and a set of guarantees that it provides upon successful completion. In this proposal, the authors recommend that compiler implementations offer switches to select a level of contract checking: on, off, pre-condition only, post-condition only. Moreover, the LLVM/Clang compiler itself supplies switches to turn on different verification tools, like the address or the thread sanitizer functionality [17] to report faulty memory accesses and race conditions.

Our idea here is to change the compiled code to provide the ability to replace specified functions with their designated test doubles upon a compiler switch. The test code of our previous example with the `Entity` and `mutex` might look like this:

```
#include "Entity.hpp"

bool try_lock_result;
bool fake_mutex_try_lock(std::mutex* self) { return try_lock_result; }

TEST_F(Fixture, Mutex) {
    SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
    Entity e;
    try_lock_result = false;
    EXPECT_EQ(e.process(1), -1);
    try_lock_result = true;
    EXPECT_EQ(e.process(1), 1);
}
```

With the `SUBSTITUTE` macro, we simply replace the `try_lock` member function of `std::mutex` with a free function named `fake_mutex_try_lock`. The first parameter of this free function holds a pointer to the object on which the original `try_lock` member function has been called.

We have to compile the test binary with the appropriate compiler flag that enables this kind of instrumentation. In our proof-of-concept implementation [21] we use the `-fsanitize=mock` switch with the LLVM/Clang compiler for this purpose. The production code shall be compiled without this flag. Actually, the idea is quite similar to Clang's thread sanitizer. Thread sanitizer is a race condition detector. It instruments each and every memory load and store so it can update some bookkeeping information to aid the detection [39].

By switching on the `-fsanitize=mock` flag we instruct the compiler to replace each and every function call expression with the following pseudo code (let us suppose that the callee is the `foo` function):

```
char* funptr = __fake_hook(&foo);
if (funptr) {
    funptr(args...);
} else {
    foo(args...);
}
```

The return value of the call to `__fake_hook` is a function pointer to the test double which will be called instead of the original function. If that return value is a `nullptr`, then the original function shall not be replaced. If the callee does return anything other than `void`, then the expression is transformed in the following way:

```
char* funptr = __fake_hook(&foo);
auto ret = result_of(&foo);
if (funptr) {
    ret = funptr(args...);
} else {
    ret = foo(args...);
}
return ret;
```

The `__fake_hook` function is defined in a shared library that we have to link to the test binary. Its implementation searches for the function pointer which has been set up by the `SUBSTITUTE` macro. Our realization uses a simple hash map to store the pointers for the original and the test double functions.

It is possible to call the replaced function from the test double. For this purpose, we need to mark the test double with a special attribute so as to avoid its instrumentation. Otherwise, the call site of the original function inside the test double would also be substituted and lead to an infinite recursion.

It should be mentioned that we can replace a callee function in a call expression only if the translation unit which holds that particular call expression has been compiled with the special flag. Quite often, the call expression where we want to substitute the callee is in the source code of the unit under test; i.e it is in the source of our active project. Thus, in most cases we do not have to recompile dependency libraries. We have to recompile dependency libraries only if we wish to substitute a non-inline callee in the library code itself.

Namespaces and member functions are handled similarly to regular free functions. Despite of the fact that member function pointers are really different from pointers to free functions, internally the compiler assigns a unique identifier to all member functions (the address of the function).

A `constexpr` [12, 5.19] function cannot be replaced with this method when it is used in a compile-time expression. However, it can be replaced whenever it is used within a runtime context since this is a runtime instrumentation.

GCC and Clang do not inline any functions when not optimizing unless we specify the `always_inline` attribute for the function [8]. Always inline functions are inlined even when the `-fno-inline` compiler switch is present [9]. Our instrumentation forces the compiler to emit the code of the specific function even when it is explicitly marked with the `always_inline` attribute; hence we can replace inline and always inline functions as well.

3.1 Evaluation

We implemented the above-presented method based on the LLVM/Clang compiler infrastructure. The modified C++ compiler and the corresponding runtime library is publicly available online [22, 21]. We measured the performance of the compilation process itself by compiling its own source code with and without the instrumentation enabled. In both cases, we started a timer to evaluate the wall clock time of the build process and we could not discover any significant difference between the two results. This is quite similar to the compile-time performance results of other sanitizers (e.g. the thread sanitizer).

We found in practice that the runtime performance of the generated binary is slowed down by a factor of 5 to 60. It depends on the number of inlined functions and call expressions. Inlining is an optimization that is turned off by this instrumentation. Our method costs a lookup and a branching for each and every call whether it is substituted or not. Our evaluation demonstrates that most of the reduction in the performance is caused by the lookup, hence it is an important future work to improve it. We shall use a shadow memory [39] (an offset address in the program's virtual memory) to access the test double's address. This way, we could achieve a similar performance to the thread sanitizer runtime performance, which is a slow down by a factor of 5 to 15.

By using our new instrumentation technique, we can write tests without intrusively changing the original unit itself. We do not have to add a new constructor to setup a dependency, and we do not have to add a new getter. Since we do not have to change the original unit at all, the production code should have the same performance as it had before adding tests. Furthermore, this method works with inline and always inline functions. In contrast to link seams, it is really obvious to identify the use of this seam (via the SUBSTITUTE macro in the test code).

4 Access Private Members

Non-intrusive testing requires the ability to access private data as well. Imagine a situation where a replaced member function has to access the internal state of the object to be able to assert on that. For instance, we may wish to check that the `mutex` is unlocked when the `try_lock` member function is called:

```
bool fake_mutex_try_lock(std::mutex* self) {
  EXPECT_EQ(self->locked, true);
}

TEST_F(FooFixture, Mutex) {
    SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
    // ... as before
}
```

There are various existing methods available for accessing private members in C++, but all have certain drawbacks. In this section we overview the existing approaches and we introduce two new procedures that attempt to overcome the difficulties.

4.1 Access via a shared reference or pointer

In this case, the unit that we wish to test has a constructor or a setter function with a reference or a pointer parameter through which we can inject the dependency. An example is:

```
Entity(Mutex& m) : m(m) {}
```

We cannot use a `unique_ptr` this way, since we need to access the dependency from the test as well; however, `unique_ptr` provides exclusive ownership only for the owner.

4.2 Access via getter

As we saw previously, it might be more natural to use a getter when the unit has exclusive ownership:

```
// Use only in tests !
Mutex& getMutex() { return m; }
```

The disadvantage here is that it violates encapsulation, i.e. it exposes an internal member.

4.3 Use preprocessor and getter

To protect the internal member, it is possible to define the getter function only when the unit is built for testing.

```
#ifdef TEST
  Mutex& getMutex() { return m; }
#endif
```

This requires support from the build system, so during the compilation of each translation unit of the test executable this flag needs to be defined. Also, test specific preprocessing is hard coded, which makes it more difficult to see through the unit's overall structure. Though it is quite obvious that the getter is used only for testing, this is actually a benefit.

4.4 Change the access level with the preprocessor

A frequently applied trick is to use the preprocessor to access private members:

```
#define private public
#include "Unit.h"
#undef private
// Test code comes from here
```

Although the standard forbids us from redefining keywords [12, 17.6.4.3.1 Macro names/2], most preprocessors accept this. The obvious drawbacks are easy to see however. All the other classes that are directly or indirectly included from `Unit.h` now expose all their internals. This opens the possibility for errors in the test code via accidentally accessing members of the dependencies, which we shall not know about (violating encapsulation).

Also, this approach does not always work. To see this, consider the following `class`:

```
class X { int a; };
```

The default access specifier is `private` in the case of C++ classes, therefore the define directive has no effect. Still, this can be circumvented with an additional define: `#define class struct`.

What is more, this is undefined behaviour because the C++ standard specifies that the order of allocation of non-static data members with different access control is unspecified [12, 9.2 Class members/13].

4.5 Friend function or class

In C++, friends have right to access private and protected members. With this approach, we declare a concrete test function inside the unit to be a friend:

```
class Entity {
public:
  friend void testClient(Entity& e);
  Entity(std::unique_ptr<Mutex> m) : m(std::move(m)) {}
  int process(int i) { if(m->try_lock()) { ... } else { ... } }
  //...
private:
  std::unique_ptr<Mutex> m;
  //...
};
void testClient(Entity& e) {
  // access e.m here
}
```

We can also declare an in-between class to be a friend; then in the tests we can use the different member functions of the friend class to access the private members.

4.6 Access via explicit instantiation

So far we have considered only intrusive methods to access private members. But there are other approaches which do not require intrusive changes in the unit we wish to test. In this section, we present an interesting non-intrusive technique then in the forthcoming subsection we present our new solution for accessing private members as a generalization of this technique.

We can access outside of the declaring class any private member if we exploit the fact that C++ allows us to pass the address of a private member in explicit instantiation [12, 14.7.2 Explicit instantiation/12]. The standard permits this behaviour, because otherwise specializing traits for private types would not be possible. Besides private members, we can access private `static` variables and functions as well with this technique.

To understand how we can exploit this fact, consider the following class:

```
class A { static int i; };
int A::i = 42;
```

We would like to access the static private variable `i`. Normally, accessing that private variable results in a compiler error:

```
int x = A::i; // Error, i is private
```

Yet, there is an exceptional case, namely when we provide a template argument in an explicit template specialization. Let us assume that we have a class template defined, so we can explicitly specialize that:

```
template struct private_access<&A::i>;
```

The template argument `&A::i` has a compile-time available value and it has the type `int*`. In this context, `&A::i` is a completely valid expression, which has the address of the private variable as the value. We need to expose this address somehow, so we define the class template `private_access` as follows:

```
template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};
```

The template parameter of `private_access` is a non-type template parameter, which is a pointer value of type `int*` known at compile-time. We define the `get()` function to return the actual compile-time value of this template parameter. It returns the address of the private static variable, since the template is instantiated with that value as an argument. By defining the `get()` function as a friend it becomes part of the enclosing namespace scope. Even so, its name is not found by normal lookup (qualified or unqualified) [12, 7.3.1.2 Namespace member definitions/3]. Therefore, we need to provide an additional *declaration* outside of the class:

```
int* get();
```

Putting this all together, our code with a usage example is the following:

```
class A { static int i; };
int A::i = 42;

template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};

int* get();

template struct private_access<&A::i>;

void usage() {
  int* i = get();
  assert(*i == 42);
}
```

Accessing private, non-static members is quite similar:

```
1  class A { int i = 42; };
2
3  template<int A::* PtrValue> struct private_access {
4      friend int A::* get() { return PtrValue; }
5  };
6
7  int A::* get();
8
9  template struct private_access<&A::i>;
10
11 void usage() {
12     A a;
13     int A::* ip = get();
14     int& i = a.*ip;
15     assert(i == 42);
16 }
```

The only difference is in the type of the template argument, which is now `int A::*`, a pointer to member. Values of this type may be pointers to any `int` data member of the class `A`. Once we get the pointer to the member in line 13, we can bind this pointer to an object, and this way, we get a reference to the data member (in line 14).

4.6.1 Generalized private access

As for our contribution, we generalized the above-presented techniques. We have created a library which automates the generation of the helper constructs to access private data members and to call private member functions (both static and non-static) [20]. So accessing private data members becomes straightforward:

```
class A { int m_i = 3; };

ACCESS_PRIVATE_FIELD(A, int, m_i)

void foo() {
  A a;
  auto &i = access_private::m_i(a);
  assert(i == 3);
}
```

Similarly, calling private functions can be achieved like so:

```
class A {
  int m_f(int p) { return 14 * p; }
};

ACCESS_PRIVATE_FUN(A, int(int), m_f)

void foo() {
  A a;
  int p = 3;
  auto res = call_private::m_f(a, p);
  assert(res == 42);
}
```

We deliberately do not use pointer-to-members in the public interface of this macro library. We think that their use is just an implementation detail that we do not wish to expose to the user.

As a first design decision, we place all components of this library into an unnamed namespace to prevent multiple definition linker errors. For instance, we want the following 3 files (`a.hpp`, `x.cpp`, `y.cpp`) to be linkable into an executable file:

```
// a.hpp
class A { int m_i = 3; };


// x.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)


// y.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)
int main() { return 0; }
```

Then we commence with the generic definition of `private_access`. We use the nested namespace `private_access_detail` as a safeguard, because we wish to avoid name clashing as the user code might have additional names defined in unnamed namespace:

```
namespace {
  namespace private_access_detail {
    template <typename PtrType, PtrType PtrValue, typename TagType>
    struct private_access {
      friend PtrType get(TagType) { return PtrValue; }
    };
  } // namespace private_access_detail
} // namespace
```

By introducing the `PtrType` type template parameter, we generalize the type of the pointer we wish to use. This might be `int*` or `int A::*` if we take our examples from the previous section. We also bring in the `TagType` type template parameter, which we use to define different instances of the `get()` function. This is achieved implicitly by instantiating the `private_access` class template with different concrete tag types.

Next, we define some helper macros for concatenation:

```
#define PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y) x##y
#define PRIVATE_ACCESS_DETAIL_CONCATENATE(x, y)                    \
  PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y)
```

We use the `PRIVATE_ACCESS_DETAIL` prefix for all the implementation macros that are supposed to be hidden from the clients of this macro library.

Afterwards, we introduce a macro which contains all those things that are common in the implementation of accessing a static or a non-static member:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name, \
2                                               PtrTypeKind)            \
3    namespace {                                                        \
4      namespace private_access_detail {                                \
5        struct Tag {};                                                 \
6        /* Explicit instantiation */                                  \
7        template struct private_access<decltype(&Class::Name),         \
8                                       &Class::Name, Tag>;             \
9        /* Define the PtrType alias */                                \
10       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_, Tag) = Type;   \
11       using PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) =       \
12           PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_,                  \
13                                             Tag) PtrTypeKind;        \
14       /* declare the get() function */                              \
15       PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) get(Tag);     \
16     }                                                                \
17   }
```

The macro parameter `Tag` is the name of the tag class we want to define and we also use it as a suffix for the name of the type aliases. `Class` denotes the qualified or unqualified name of the class we wish to provide access to. `Type` is the type of the member variable. The parameter `PtrTypeKind` describes what kind of pointer are we dealing with, namely a simple pointer or a pointer-to-member. For instance, it may have the strings `*` or `A::*`. First, we define the tag type (line 5), then comes the explicit instantiation with the type and address of the member and with the recently defined tag type (line 7-8).

Then, we define a type alias (with `PtrType_` prefix) for the concrete type of the pointer (line 9-13). Basically, this alias is formed from the concatenation of the `Type` and `PtrTypeKind` parameters. For example, in the case of a pointer-to-member, the canonical type of the type alias might be `int A::*`. The twist here is

that we need to add two type aliases, because pointer-to-member-functions cannot be expressed generically with one alias, e.g.:

```
using PtrType1 = int(int) *; // ERROR
using Alias = int(int);
using PtrType2 = Alias *; // OK
```

Next, we declare the `get()` function to make it available for finding by normal name lookup (line 15).

Following this, we define the specific macro for non-static member fields.

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(Tag, Class, Type, \
2                                                     Name)             \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace access_private {                                       \
7        Type &Name(Class &&t) {                                        \
8          return t.*get(private_access_detail::Tag{});                 \
9        }                                                              \
10       Type &Name(Class &t) {                                         \
11         return t.*get(private_access_detail::Tag{});                 \
12       }                                                              \
13       using PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag) = Type;        \
14       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) =              \
15           const PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag);           \
16       PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) &                    \
17           Name(const Class &t) {                                     \
18         return t.*get(private_access_detail::Tag{});                 \
19       }                                                              \
20     }                                                                \
21   }
```

The macro parameters `Tag`, `Class`, `Type` and `Name` have the exact same meanings as before. In line 3, we call the previously described macro to generate all the generic code we need. We pass "Class::*" as a macro argument, since we are dealing with non-static members. If we were dealing with static members, then the argument would be "*". Then, we define two overloaded functions in the enclosing `access_private` namespace with the name which is equal to the name of the member we are exposing, e.g. "m_i" (line 7-12). These overloads are for those cases where the class instance is bound to an rvalue reference or to a non-const lvalue reference. We bind the result of `get()` function to the object of the class; and then we return with a reference to the resulting member. Later, (in lines 13-19) we add another overload that handles the cases where the object is bound to a const lvalue reference. In this case we would like to preserve the constness of the object, therefore we should return with a const reference to the member. So, we create a type alias for this const reference type (lines 13-15). Here, once again we need to use two type aliases because we would like to avoid warnings that arise from duplicated const qualifiers. If we used just one alias, then we would get this warning if the `Type` macro parameter already contains a const qualifier. After defining the type alias, we use this in the definition of the third overload (lines 16-19).

The implementation of accessing static fields is very similar to the implementation of accessing non-static members.

The realization of calling private member functions, however, requires an explanation:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(Tag, Class, Type,   \
2                                                   Name)               \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace call_private {                                         \
7        template <typename Obj,                                        \
8                  std::enable_if_t<std::is_same<                       \
9                      std::remove_reference_t<Obj>, Class>::value> * = \
10                     nullptr,                                         \
11                 typename... Args>                                    \
12       auto Name(Obj &&o, Args &&... args) -> decltype(             \
13           (std::forward<Obj>(o).*get(private_access_detail::Tag{}))( \
14               std::forward<Args>(args)...)) {                      \
15         return (std::forward<Obj>(o).*                             \
16                 get(private_access_detail::Tag{}))(                \
17             std::forward<Args>(args)...);                          \
18       }                                                            \
19     }                                                              \
20   }
```

Here, we again call the common macro that does the explicit instantiation (lines 3-4). Then we perfect forward both the object and the parameters of the private function we wish to call (lines 7-17). We bind the pointer-to-member-function (result of the `get()` function) to the perfect forwarded object and then we call the resulting member function with the forwarded arguments (lines 15-17). We use the same expression's type as the trailing return type in the header of the function (lines 12-14). (Note that in C++14 there is no need to specify the trailing return type.) We also restrict the set of function template instantiations that can participate in the overload resolution with the `enable_if`. The goal here is to exclude a template function when the type of the object is different from the type of the `Class` parameter. By doing this, we get a more compact error message if we misuse the library for some reason. Otherwise we would get error messages originating from the body of the function template.

The implementation of calling static member functions is very similar to the implementation of calling non-statics, but we do not need the `enable_if` there, since we do not have an object in that case.

Somehow we need to generate unique tag types, so for this we use the built-in `__COUNTER__` macro which returns an integer and is incremented by the preprocessor each time it is referenced. `__COUNTER__` is not a standard macro, but it is available on most mainstream compilers (GCC, Clang, MSVC).

```
#define PRIVATE_ACCESS_DETAIL_UNIQUE_TAG                               \
  PRIVATE_ACCESS_DETAIL_CONCATENATE(PrivateAccessTag, __COUNTER__)
```

The macro `PRIVATE_ACCESS_DETAIL_UNIQUE_TAG()` will generate a unique name with the prefix `PrivateAccessTag`. Finally, we can define the main macros of the library with the help of the unique tag generator:

```
#define ACCESS_PRIVATE_FIELD(Class, Type, Name)                        \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(                          \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)
```

```
#define ACCESS_PRIVATE_FUN(Class, Type, Name)                          \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(                            \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)
```

During the compilation of one translation unit, each invocation of these macros generates different tag types. Since these tag types are defined in an unnamed namespace, we will not have any linkage errors of duplicate symbols when linking multiple translation units together.

Now we have seen how we can access private member fields and how we can call private member functions, regardless of whether if they are static or not. However, this library has some limitations. We cannot access private types, because the only valid context of using that private type is inside the explicit instantiation. We cannot call private constructors nor destructors. This is because a pointer to member cannot bind to a constructor (since we do not have the object unless the constructor is called). Nor can a pointer to member bind to a destructor because there is no valid expression in C++ to grab the address of a destructor. We have a link time error in the case of in-class declared `const static` variables (without an out-of-class definition). This is because we would take the address of that variable, and if that is not defined (i.e the compiler does a compile-time insert of the const value), we would be trying to dereference an undefined symbol. Owing to all of these limitations we were motivated to come up with a more sophisticated solution.

Note that the Java language has a built in support to achieve something similar. With `setAccessible` we can indicate that the reflected object should suppress access checking when it is used [30].

4.7 Out of class friend

As we saw above, private access via explicit instantiation does not work for all kinds of private entities. So, our other contribution is to explore the idea of a new lingual element with which we would be able to access all kinds of private members. In our non-intrusive approach, we define a function or a class as `friend` out of the befriending class:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) {} else {} }
  //...
private:
  Mutex m;
  //...
};

friend for(Entity<StubMutex>) void test_try_lock_fails() {
  Entity<StubMutex> e;
  auto& m = e.m; // access the private member
  // set up try_lock result value to false and do the assertions ...
}
```

Based on the LLVM/Clang compiler (version 3.6.0) [17], we created a proof-of-concept implementation for out-of-class friends and it is now available online (see

[23]). The goal of this implementation is to demonstrate that the idea is indeed feasible, though it is not our objective to provide a full featured perfect realization. Therefore, we add some restrictions to the functionality and we do not implement proper error handling.

To ease the implementation, we use C++ attributes [12, 7.6 Attributes] instead of modifying the existing grammar. More specifically, we use the GCC `__attribute__` syntax because the standard `[[attribute]]` syntax implementation was not complete in the Clang version we used. By using attributes, we skip the problem of parsing and we can focus on the new semantic actions. So, the above definition of `test_try_lock_fails` with attributes is the following:

```
__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
```

However, prior to this definition we need to explicitly instantiate the `Entity` class template.

```
template class Entity<StubMutex>;
```

This is required because the attribute's associated semantic action attempts to access all the details of its type parameter (`Entity<StubMutex>`). In a future study, it might be possible to modify the realization so as to implicitly do the instantiation during the semantic action of the friend attribute. The instantiation could be triggered just before accessing the details of the type parameter.

The definition with the attribute behaves exactly as any other in-class defined friend definition. As such, it is not found by normal lookup unless we declare it explicitly. Of course we wish it to be found by normal lookup, otherwise we will not be able to call the function. Overall, this means that our test code should have the form:

```
template class Entity<StubMutex>;

__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
void test_try_lock_fails();

// part of the test framework
void testDriver() {
  test_try_lock_fails();
  // ... call other test functions
}
```

Here `testDriver` is the function which embodies the test framework, whose task is to execute each test case (or test suite) one-by-one. Another restriction of this particular realization of out-of-class friends is to allow only functions to be declared friends in this way.

After defining the constraints of such an attribute-based implementation we can explore the concrete realization steps. First, we define our new attribute in Clang's `Attr.td` file:

```
def OutOfClassFriend : InheritableAttr {
  let Spellings = [GCC<"friend">];
  let Args = [TypeArgument<"Host">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented];
}
```

`Spellings` defines the list of the supported attribute syntaxes, but this time it is only the GCC style. The attribute syntax also defines the name of the attribute, in our case it is `friend`. `Args` specifies the list of the attribute arguments. Our friend attribute has only one argument which is a type. This type argument refers to the type that we would like to be the host class (the befriending class), i.e. the class for which we define the additional friend function. `Subobjects` describes the list of the lingual elements that might have this attribute. In this case, we only allow functions to have it. Note that implementing this attribute for classes is an important issue for future research.

Once we have the attribution definition in place, the Clang infrastructure will generate all the necessary parsing code. What is left is for us to define the semantic action for the new attribute and to hook that action into the existing complier machinery. As for the hooking, we need to add a new function call in the `ProcessDeclAttribute` function. This function is dedicated to apply a specific attribute to the specified declaration if the attribute applies to declarations. (Our attribute applies to function declarations.)

```
static void ProcessDeclAttribute(Sema &S, Scope *scope, Decl *D,
                                 const AttributeList &Attr,
                                 bool IncludeCXX11Attributes) {
  //...
  case AttributeList::AT_OutOfClassFriend:
    handleOutOfClassFriendAttr(S, D, Attr);
    break;
  //...
}
```

The semantic action for the new attribute is defined as follows:

```
1  static void handleOutOfClassFriendAttr(Sema &S, Decl *D,
2                                         const AttributeList &Attr) {
3    // Get the attribute type argument as QualType
4    ParsedType PT;
5    if (Attr.hasParsedType())
6      PT = Attr.getTypeArg();
7    else { // TODO error handling
8    }
9    TypeSourceInfo *QTLoc = nullptr;
10   QualType QT = S.GetTypeFromParser(PT, &QTLoc);
11   if (!QTLoc)
12     QTLoc = S.Context.getTrivialTypeSourceInfo(QT, Attr.getLoc());
13
14   // The type argument must be a CXXRecordDecl
15   RecordDecl *RD = getRecordDecl(QT);
16   assert(RD);
17   CXXRecordDecl *CRD = cast<CXXRecordDecl>(RD);
18   // The attribute is subject of a FunctionDecl
19   FunctionDecl *FD = cast<FunctionDecl>(D);
20   // Set this function as a friend function
21   FD->setObjectOfFriendDecl();
22   // Create a new friend decl for the befriending class
```

```
23    FriendDecl::Create(S.Context, CRD, D->getLocation(),
24                       cast<NamedDecl>(D), Attr.getLoc());
25    // For the record, Add the attribute to the Decl
26    D->addAttr(::new (S.Context) OutOfClassFriendAttr(
27        Attr.getRange(), S.Context, QTLoc,
28        Attr.getAttributeSpellingListIndex()));
29  }
```

The `S` parameter holds a reference of the monumental `Sema` class which is responsible for semantic analysis and AST building in the Clang compiler. The `D` parameter represents the declaration which has the attribute. The attribute itself is described with the `Attr` parameter. The first step is to get the type parameter of the attribute as a `QualType` (line 3-12). A `QualType` holds the basic type (e.g. `int`) and all the qualifiers – if any – on that type. For this, we get the `ParsedType` from the `Attr` with the `getTypeArg()` function (lines 4-6). A `ParsedType` is an opaque pointer for `QualType`s, i.e. this is a type erased generic holder, this is something similar to `void*`. Next, we get the underlying `QualType` from the `ParsedType` and we set the location of it (lines 10-12). If we cannot get the location information for the type, we simple set it to the location of the attribute (lines 11-12).

Afterwards, we get the `RecordDecl` instance from the `QualType` instance with the help of the `getRecordDecl` function (line 15). This function returns a null pointer if the `QualType` does not represent a record declaration. In Clang, a `RecordDecl` is the type of the AST node that is created for C structs and unions. Similarly, a `CXXRecordDecl` is specifically for C++ classes, structs and unions. This means that we can safely cast the record declaration to a `CXXRecordDecl` (line 17). The cast expression used here is a Clang specific cast, which is a "checked cast" operation. It converts a pointer or reference from a base class to a derived class, causing an assertion failure if it is not really an instance of the right type [16].

Next, we get the more specific function declaration (`FunctionDecl`) from the parameter `Decl` (line 19). The conversion from `Decl` to `FunctionDecl` must succeed, since we explicitly specified in the `Attr.td` file that this attribute is valid only for function declarations. So we use the checked cast again. Then we set up this function declaration as a friend declaration (line 21).

Later, we create the AST node for this new friend declaration (lines 23-24). This friend declaration references the previously synthesized `CRD` pointer as the befriending class, and the `D` parameter as the friend declaration. Once we have the friend declaration in place, the access checking mechanism will assess the target function like any other regular friend function.

As a last step, we register the attribute for the declaration (lines 26-28). This step is not essential, but it makes the whole procedure complete. We did this because in some future static analysis or other tool might want to process this information.

5 Related Work

There are many unit test frameworks available for C++ [40, 10, 32]. These frameworks provide only the basic tooling for creating test suites, test cases and assertions within the test cases. They do not provide any device for creating tests for legacy code without refactoring.

The experimental Boost DI framework places an emphasis to ease the creation of object trees [14]. However this framework requires that all the dependencies be injected via a constructor, i.e. we should refactor our legacy code intrusively in order to use it.

Non-intrusive testing can be done by the means of certain seams. Seams were introduced by Feathers [4]. In his book, he describes what a seam is in the context of legacy object-oriented code and he defines the three basic seams, these being preprocessor, link and object seams. He does not focus on C++, for instance he explains link seams with Java and classpath settings. He presents the refactoring method called *Extract Interface* for breaking dependencies.

Rüegg and Sommerlad elaborated the concept of seams in C++ [35]. They added a new seam for C++, called the *compile seam*. Their study expanded on the advantages, disadvantages and usability of the four seams. They presented three different techniques for linker seams, namely:

- shadow functions through linking order (original function cannot be called),
- wrapping functions with GNU's linker `wrap` command line option (the original function can be called),
- runtime function interception with `LD_PRELOAD` (the original function can be called and does not require relinking).

They also created a refactoring tool which is a plug-in for the Eclipse CDT platform including a C++ based mock object library [28]. Their tool supports refactoring towards all four mentioned C++ seams. Refactoring towards compile seams is possible via the technique they call the *Extract Template Parameter*.

Accessing private *non-static* data members via static pointers was first presented by Johannes Schaub [36]. Later it was extended by Chandra Shekhar Kumar [15] so as to use friend functions instead of static pointers. We also extended Kumar's approach to make it simpler and cleaner and we based our generic macro library implementation on the simplified version. To the best of our knowledge, accessing private *static* variables had never been presented before.

6 Vision/Future

With the help of compile-time reflection it would be possible to not change the `Entity` only for the simple reason that we wish to test it.

```
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  std::mutex m;
  //...
};

void testClient() {
  using EntityUnderTest =
    test::ReplaceMemberType<Entity, std::mutex, StubMutex>;
  EntityUnderTest e;
  auto& m = e.get<StubMutex>();
  // Test code as before
}
```

Here, `EntityUnderTest` is a type alias to such a type, which is equivalent to the `Entity` type except that all of its members with type `std::mutex` are replaced by the `StubMutex` type. Also, this type could give access to its internal mutex instance via its `get` function template. The above code requires the language capability of being able to declare variables and functions based on reflected names and types. Unfortunately, current C++ compile-time reflection proposals do not handle this language capability [42, 24]. This kind of reflection is sometimes called intercession (aka reification). For this technique to work, the given class has to be header-only, because the compiler has to know its internal layout and types to be able to replace some of them with another type. This header-only requirement might sound frightening, but with C++ modules it might be less painful to use header only classes [33].

7 Conclusion

The interweaving of the original code with the test code is an everyday problem in C++. Testing object-oriented systems frequently requires the replacement of objects representing the state of unit under test, either for exercise them or to mock them with a test double. While in managed programming languages dependency injection is supported by language features and libraries, in C++ the programmer often has to apply techniques which spoil the program structure, weaken encapsulation, and degrade performance.

In this paper, we overviewed the most frequent seams and their enabling points to alter the original behaviour of C++ programs for test purposes. We discussed the detrimental effects they may cause in the code structure and their compile-time and run-time performance.

To overcome these difficulties, we decided to introduce a new, non-intrusive compiler instrumentation-based approach for dependency replacement. The compiler collects information on the designated (member)functions at compile-time and creates a hook to make it possible to replace the called function during run-time. This technique could be used even in the case of inline functions and header only classes too, in a way that test code could be independent entirely from the unit

under test. The idea may be viewed as a new kind of seam that avoids most of the problems with the existing seams. We implemented this method as a patch for the LLVM/Clang compiler infrastructure, and evaluated its compile-time and run-time performance.

All non-intrusive testing methods require access to the internal state of the objects under test. Our method is of course no exception. Therefore, for the sake of accessing private members we discussed different techniques available for C++. Exploiting an exceptional language rule concerning explicit template instantiation provides an interesting way of accessing private non-static data members. We generalized the technique to access static members and member functions as well. Then we created a library to automate the access.

Since our new technique above had still some shortages, we presented a more generic method to access private or protected members. Friend declarations added outside of a class could provide a full, non-intrusive solution to separate test related code from the source of the unit under test. This new language element has the capability to work on every private assets, data, function, or type. We also realized a prototype based on C++ attributes to demonstrate the feasibility of the idea.

The out-of-class friends together with the compiler instrumented dependency replacement solution not only prevent the degradation of the code structure, but also avoid performance penalties. In a future study we would like to replace whole types based on the future standardized compile-time reflection of C++.

References

[1] Bertolino, Antonia and Marchetti, Eda. 1 a brief essay on software testing, 2004.

[2] Bright, Walter. C++ compilation speed, August 2010. `http://www.drdobbs.com/cpp/c-compilation-speed/228701711`.

[3] Driesen, Karel and Hölzle, Urs. The Direct Cost of Virtual Function Calls in C++. In Anderson, Lougie and Coplien, James, editors, *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 306–323. ACM, 1996.

[4] Feathers, Michael. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[5] Fowler, Martin. The new methodology. `http://www.martinfowler.com/articles/newMethodology.html`.

[6] Fowler, Martin. Using a service locator. `http://martinfowler.com/articles/injection.html\#UsingAServiceLocator`.

[7] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] gcc.gnu.org. An inline function is as fast as a macro, 2016. https://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[9] gcc.gnu.org. Options that control optimization, 2016. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] Google. Google test. https://github.com/google/googletest.

[11] ISO. ISO C standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[12] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2014.

[13] Khan, Mohd. Ehmer and Khan, Farmeena. Importance of software testing in software development life cycle, 2014.

[14] Krzysztof, Jusiak. [boost].di. http://boost-experimental.github.io/di/index.html.

[15] Kumar, C.S. *Advanced C++ Faqs: Volumes 1 & 2*. Createspace Independent Pub, 2014.

[16] llvm.org. Llvm programmer's manual. http://releases.llvm.org/3.6.0/docs/ProgrammersManual.html.

[17] llvm.org. clang: a C language family frontend for llvm, 2016. http://clang.llvm.org.

[18] Majchrzak, Tim A. *Improving Software Testing: Technical and Organizational Developments*. Springer Publishing Company, Incorporated, 2012.

[19] Mane, Dashrath, Ojha, Namrata, and Chitnis, Ketaki. The spring framework: An open source java platform for developing robust java applications. *International Journal of Innovative Technology and Exploring Engineering*.

[20] Márton, Gábor. Access private, 2016. https://goo.gl/ynaZv5 https://github.com/martong/access_private.

[21] Márton, Gábor. Instrumentation for testing, 2016. https://goo.gl/FmNT5Jhttps://github.com/martong/finstrument_mock.

[22] Márton, Gábor. Modified Clang++ for intrumentation for testing, 2016. https://goo.gl/zutxKchttps://github.com/martong/clang/tree/finstrument_mock.

[23] Márton, Gábor. Out-of-class friend, 2016. https://goo.gl/kJgnzGhttps://github.com/martong/clang/tree/out-of-class_friend_attr.

[24] Márton, Gábor and Porkoláb, Zoltán. C++ compile-time reflection and mock objects. *Studia Univ. Babes–Bolyai Informatica*, LIX(2), 2014.

[25] Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. Item 41, Understand implicit interfaces and compile-time polymorphism.

[26] Meyers, Scott. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[27] Mihalicza, József, Porkoláb, Zoltán, and Gabor, Abel. Type-preserving heap profiler for C++. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 457–466. IEEE Computer Society, 2011.

[28] mockator.com. An eclipse cdt plug-in for c++ seams and mock objects. `http://mockator.com/`.

[29] Network, Microsoft Developer. Unity container. `https://goo.gl/YQwWlE` `https://msdn.microsoft.com/en-us/library/ff647202.aspx`.

[30] Oracle. Java Platform, Standard Edition 6 API Specification, Class AccessibleObject. `https://goo.gl/rfKFRA` `https://docs.oracle.com/javase/6/docs/-/api/java/lang/reflect/AccessibleObject.html`.

[31] Parent, Sean. Inheritance is the base class of evil, 2013.

[32] Peter, Sommerlad and Emanuel, Graf. Cute: C++ unit testing easier. In *OOPSLA '07: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 783–784, New York, NY, USA, 2007. ACM. 548071.

[33] Reis, Gabriel Dos. A module system for C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf`.

[34] Reis, Gabriel Dos, Daniel García, J., and Logozzo, Francesco. Simple contracts for C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4415.pdf`.

[35] Rüegg, Michael and Sommerlad, Peter. Refactoring towards Seams in C++. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 117–123, Piscataway, NJ, USA, 2012. IEEE Press.

[36] Schaub, Johannes. Access to private members: Safer nastiness. `https://goo.gl/aG6HEv` `http://bloglitb.blogspot.hu/2010/07/access-to-private-members-thats-easy.html`.

[37] Schwarz, Niko, Lungu, Mircea, and Nierstrasz, Oscar. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *Journal of Object Technology*, 11(1):3:1–23, April 2012.

[38] Seemann, Mark. *Dependency Injection in .NET*. Manning, 2011.

[39] Serebryany, Konstantin and Iskhodzhanov, Timur. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[40] sourceforge.net. CppUnit – C++ port of JUnit. `https://sourceforge.net/projects/cppunit/`.

[41] Stroustrup, Bjarne, Sutter, Herb, et al. C++ core guidelines, 2016. `https://goo.gl/7ZXiov` `https://github.com/isocpp/-/CppCoreGuidelines/blob/master/CppCoreGuidelines.md\#Rr-scoped`.

[42] Tomazos, Andrew and Kaeser, Christian. Type property queries. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4428.pdf`.

[43] wikipedia.org. Dependency injection. `https://goo.gl/OjlpOY` `http://en.wikipedia.org/wiki/Dependency_injection`.

[44] wikipedia.org. Service locator pattern. `https://goo.gl/1KFxKj` `https://en.wikipedia.org/wiki/Service_locator_pattern`.

[45] Winters, T. and Wright, H. All your tests are terrible, 2015.

Acta Cybernetica **23** (2017) 687–713.

Evaluating Well-Formedness Constraints on Incomplete Models

Oszkár Semeráth* and Dániel Varró*

Abstract

In modern modeling tools used for model-driven development, the validation of several well-formedness constraints is continuously been carried out by exploiting advanced graph query engines to highlight conceptual design flaws. However, while models are still under development, they are frequently partial and incomplete. Validating constraints on incomplete, partial models may identify a large number of irrelevant problems. By switching off the validation of these constraints, one may fail to reveal problematic cases which are difficult to correct when the model becomes sufficiently detailed.

Here, we propose a novel validation technique for evaluating well-formedness constraints on incomplete, partial models with *may* and *must* semantics, e.g. a constraint without a valid match is satisfiable if there is a completion of the partial model that may satisfy it. To this end, we map the problem of constraint evaluation over partial models into regular graph pattern matching over complete models by semantically equivalent rewrites of graph queries.

Keywords: partial models, model validation, graph patterns

1 Introduction

Context In Model-Driven Development (MDD), models are the main design artifacts, from which documentation, system configuration, or even source code can be automatically generated. MDD is widely used in industry in various domains including business modeling, avionics and automotive [38] as it provides early validation and advanced automation. When developing complex systems, multiple design rules and well-formedness constraints have to be checked repeatedly over large (graph) models [4] in order to ensure the validity of models throughout the entire design process starting from an early stage of design.

During development, the level of uncertainty represented in the models gradually decreases until all critical design decisions have been made. However, certain constraints can only be checked at the right level of abstraction, i.e. after some

*Budapest University of Technology and Economics Department of Measurement and Information Systems, MTA-BME Lendület Research Group on Cyber-Physical Systems, McGill University, Department of Electrical and Computer Engineering. E-mail: {semerath,varro}@mit.bme.hu

DOI: 10.14232/actacyb.23.2.2017.15

design decisions have already been made. When a new constraint is violated, engineers may need to rethink some parts of the system and reiterate on some previous design decisions taken earlier. The uncertainty, which is inherently present in high-level initial models, make design decisions drawn from them especially risky.

Problem statement Partial models have been introduced in [14] to formally capture uncertainty in various design phases. Existing techniques for partial models allow a modeler to explicitly express model uncertainty, or assess possible design candidates [31], but they do not provide sufficient support for the evaluation of well-formedness constraints over partial models. While there is efficient tool support for defining and checking well-formedness constraints and design rules over regular model instances by using graph pattern matching, the evaluation of the same constraints have only been addressed by SMT/SAT solving tools, which have major scalability problems as the size of the models starts to grow.

Objective Our goal is to evaluate well-formedness constraints over incomplete, partial models by graph pattern matching instead of SAT/SMT solving. The key conceptual challenge is that while existing model query and transformation tools evaluate graph constraints over models with closed world semantics (i.e. the model is assumed to be complete), evaluating constraints on partial models necessitates an open world semantics, as new elements may be added to the model later on, which may satisfy (or violate) the constraints. Instead of proposing dedicated graph pattern matching algorithm that operates over partial models, we rewrite the original graph constraints (to be matched with open world semantics over partial models) into an equivalent constraint (to be matched with traditional closed world semantics over regular models).

Contribution Here we present a novel technique for evaluating well-formedness constraints on partial models. Our technique uses the partial snapshots [32] to represent incomplete partial models, and it is compatible with EMF (Eclipse Modeling Framework) [36] which is the de facto industrial modeling standard in MDD. Well-formedness constraints are captured as graph queries using the pattern language of VIATRA [4]. From an input graph query, our approach generates an extended, but semantically equivalent graph query.

Added value Using this technique, design rules specified for concrete models can be automatically checked for incomplete, unfinished models to (i) detect invalid elements, and (ii) identify invalid design options during the development. Additionally, this technique can be used to (iii) enumerate all possible ways to correct currently invalid partial instance model, or (iv) list all options to inject errors by extending a currently valid model. Lastly, our approach uses approximations and an efficient graph query engine, which enables the use of this technique directly on an existing modeling environment. Performing such an analysis was unfeasible previously with logic solvers [33].

As a long term benefit, model generation techniques [33, 31] can be supported efficiently, as we can reuse existing graph pattern matching tools like the VIATRA framework to evaluate well-formedness constraints over incomplete, partial models, just as we do for for regular, complete models.

Structure of the Paper The rest of the paper is structured as follows: In Section 2 we summarize the necessary modeling concepts and introduces a case study for partial models. In Section 3 we provide an overview on how to evaluate model queries with open-world semantics, then in Section 4 we show how graph patterns can be transformed to their open-world equivalent. In Section 5 we provide comparison with other approaches in the literature. Lastly, in Section 6 we draw conclusion and make suggestions for future work.

2 Preliminaries

2.1 Motivating example: Yakindu Statecharts

Yakindu Statecharts Tools [39] is an industrial integrated modeling environment developed by Itemis AG for the development of reactive, event-driven systems based on the concept of statecharts captured in combined graphical and textual syntax. Yakindu simultaneously supports static validation of well-formedness constraints as well as the simulation of (and code generation from) statechart models. Examples in this paper are illustrated by the validation of partial Yakindu models.

The behavior model of a sample coffee machine is illustrated in Figure 1. In **Step I**, an initial statechart is developed, highlighting the key phases of the operation. Initially, the machine starts in state Ready. Then, after inserting coins, a drink can be selected in state Select. After the selection the machine start filling the coffee, and gives back the change in state Service. When the drink is ready, and the change is returned, the coffee machine goes back to the initial state.

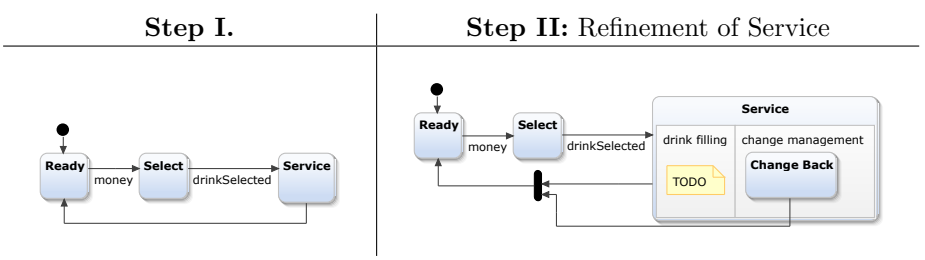


Figure 1: Example statechart model under developement.

Next, in **Step II** of Figure 1, the developer makes the first steps to refine the Service state by adding two new regions, namely one that manages the filling of the drink (drink filling), and one that manages the change (change management). When

both regions are ready, then the two regions synchronize to the initial state. At present, the model in **Step II** is only an unfinished partial model.

There are several well-formedness constraints (aka design rules) in the development environment to validate statecharts. Two examples are the following:

1. Each region needs to have exactly one entry, which has a transition to a state in the same region.
2. The target and source states of a synchronization have to be contained in the same parent state.

Both constraints are defined for complete models, and while they can be evaluated on partial models, it provides less relevant information during development:

1 The first constraint is invalid for regions drink filling and change management as they do not have initial states. However, as design decisions about internal states and entries have not yet been made, these are not real errors of the model, but a direct consequence of the incomplete model.

2 However, the synchronization in **Step II** synchronizes states that are not parallel, thus it is a design flaw that will be present in any possible completion.

2.2 Metamodels, Models, Partial Models

A metamodel defines the main concepts, relations and the basic structure of a domain-specific language (DSL). In this paper, domain models are captured by the Eclipse Modeling Framework (EMF) [36], which is frequently used in industrial modeling tools.

An extract of metamodel for Yakindu statecharts describing the state graph is illustrated in Figure 2. A state machine consists of Regions, which in turn contain states (called Vertexes) and Transitions. An abstract state Vertex is further refined into RegularStates (like State) and PseudoStates like Entry and Synchronization states. Note that we intentionally kept the generalization hierarchy unchanged and we simplified the original metamodel by by removing certain elements.

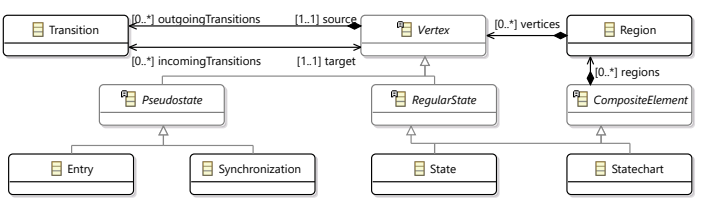


Figure 2: Simplified Yakindu state graph metamodel.

Metamodel elements are mapped to a set of logic relations as defined in [32, 20], which are revisited below:

- **Classes (CLS)**: An *EClass* captures some core concepts in a DSL. In EMF, *EClass*es can be instantiated to *EObjects*, where the set of objects of a model M is denoted by $\mathsf{Objects}_M$. If o is an instance of a type C, it is denoted by $\mathsf{C}(o)$.

- **References (REF)**: *EReference*s between classes S and T capture a binary relation $R(S,T)$ of the metamodel. When two objects o and t are in a relation R, an *EReference* is instantiated leading from o to t denoted by $\mathsf{R}(o,t)$.

- **Attributes (ATT)**: *EAttributes* enrich a class C with values of predefined primitive types like integers and strings, etc. by binary relations $A(C,V)$. If an object o stores a value v as attribute A it is denoted as $\mathsf{A}(o,v)$.

Further structural restrictions implied by a metamodel (and formalized in [32]) include (1) **Generalization (GEN)**, which expresses the fact that a more specific (child) class has every structural feature of the more general (parent) class, (2) **Type compliance (TC)** requires that for any relation $\mathsf{R}(o,t)$, its source and target objects o and t need to have compliant types, (3) **Abstract (ABS)**: If a class is defined as *abstract*, it is not allowed to have direct instances, (4) **Multiplicity (MUL)** of structural features can be limited with upper and lower bounds in the form of "lower..upper" and (5) **Inverse (INV)**, which states that two parallel references of opposite direction always occur in pairs. Finally EMF instance models are often expected to be arranged into a *containment hierarchy*, which is a directed tree along references marked in the metamodel as containment (e.g. regions or vertices).

Model M is a valid instance of a metamodel *Meta* (denoted by $M \models \mathit{Meta}$) if (i) all classes, references and attributes are defined in *Meta* and (ii) satisfies the structural constraints $(1) - (5)$ [32].

Partial models Partial (or incomplete) models arise when we are still planning to add more elements to a model, thus certain constraints can be violated. P is a partial model of a (partial or complete) model M (denoted as $P \subseteq M$) if M contains P as a submodel, so M can be created from P by adding model elements to P, Formally: $P \subseteq M$ holds if (i) P satisfies structural constraints $(1) - (4)$ above with the possible exception of lower multiplicity in **(MUL)** and (ii) there is an injective morphism $f : \mathsf{Objects}_P \to \mathsf{Objects}_M$ which satisfies the following constraints:

1. For each object o_1 and o_2: $o_1 = o_2 \Leftrightarrow f(o_1) = f(o_2)$.

2. For each class C and each object $o \in \mathsf{Objects}_P : \mathsf{C}(o) \Leftrightarrow \mathsf{C}(f(o))$.

3. For each reference R and each object pair $s,t \in \mathsf{Objects}_P : \mathsf{R}(s,t) \Rightarrow \mathsf{R}(f(s),f(t))$.

4. Finally, for each attribute type A and attribute value v and each object $o \in \mathsf{Objects}_P : \mathsf{A}(o,v) \Rightarrow \mathsf{A}(f(o),f(v))$

In a standard EMF model it is not specified which part of the model is complete, and which one is under development. Therefore, a partial model can be extended in any part: new references or attribute values can be added to any objects, or new child objects of any type can be added to the model, as long as structural constraints $(1) - (4)$ are not violated.

2.3 Graph Patterns

Well-formedness constraints are often captured by graph patterns (GP) [37, 4], which is an expressive formalism used for various purposes in model-driven development alternatively for standard OCL constraints [23]. A graph pattern is a graph-like structure representing several conditions (or constraints) matched against an instance model.

In the VIATRA pattern language, a **graph pattern** $q(p_1, \ldots, p_n) = def$ is defined by a name q and symbolic parameters $p_1, \ldots, p_n$, and constraints over the parameters (captured by *body*). A pattern may have multiple bodies with *constraints*, and may introduce additional local variables beside the parameters. The constraints available in the pattern language include:

- *Classifier constraint:* checks whether a variable is an instance of a class, i.e. checks whether $\mathsf{C}(o)$ true.
- *Path constraint:* requires a specific reference, an attribute, or a path of reference and attribute sequence between two variables.
- *Equality constraint:* specifies that two variables have to be mapped to the same model element.
- *Pattern call constraint:* enables the composition of multiple patterns. The positive pattern call refers to another pattern and specifies that the called pattern must be satisfied in the context of the actual parameters. Additionally, a pattern may be composed negatively (`neg` keyword), which means that the target negative pattern is not allowed to have a valid match along the actual parameters. Finally, it is possible to compute the transitive closure of a two-parameter pattern by the `+` symbol.
- *Count expression:* counts the number of matches of a pattern.
- *Check constraint or eval:* evaluates a specific attribute expression on the variables of the pattern and accepts matches only if the result of an attribute condition is true.

A **match** m of pattern $q(p_1, \ldots, p_n) = def$ over model M maps each symbolic parameter p_i to a model element (object, enum literal or primitive) from the target model M. A match is valid if it satisfies each constraint in a body of the pattern def: $\forall m : M \models \bigvee_{body \in def} body(m(params))$. A **partial match** is a partial function m^p where only a subset of parameters is mapped to model elements.

```
1  pattern entryInRegion(r, e) {
2    Entry(e);
3    Region.vertices(r,e);
4  }
5  pattern noEntry(r) {
6    Region(r);
7    neg find entryInRegion(r,_e);
8  }
9  pattern multipleEntry(r) {
10   find entryInRegion(r, e1);
11   find entryInRegion(r, e2);
12   e1 != e2;
13 }
```

$$\mathsf{entryInRegion}(r, e) := \mathsf{Entry}(e) \wedge \mathsf{vertices}(r, e)$$

$$\mathsf{noEntry}(r) := \forall \_e : \mathsf{Region}(r) \wedge \neg \mathsf{entryInRegion}(r, \_e)$$

$$\mathsf{multipleEntry}(r) := \exists e_1, e_2 : \mathsf{entryInRegion}(r, e_1) \wedge \mathsf{entryInRegion}(r, e_2) \wedge e_1 \neq e_2$$

Figure 3: Example validation patterns

A match of a pattern is marked by $M, m \models p$. Furthermore, we use $[M, m \models p]$ as a predicate over matches, which is evaluated to true if m is a valid match of p in M, and $\neg[M, m \models p]$ means m is not a valid match of p.

The main task of a graph query engine is to evaluate graph pattern over a model by gradually extending partial matches to a complete match. When a graph pattern is evaluated on a model, one distinguishes between zero or more matches: in the first case the query evaluates to false, while in the second case, it evaluates to true.

Example Three graph patterns are illustrated in Figure 3 along with their mathematical formalization. Pattern *entryInRegion* collects all entry nodes e of a region r, and constructed from a classifier constraint ($\mathsf{Entry}(e)$) and a path constraint ($\mathsf{vertices}(r, e)$). Pattern *noEntry* identifies regions r without an entry node by negative composition (call) to *entryInRegion* (using a negative predicate $\neg\mathsf{entryInRegion}(r, \_e)$). Lastly, pattern *multipleEntry* highlights regions r with more than a single entry node.

3 Matching Graph Queries on Incomplete Models

When evaluating a graph query over an incomplete partial model, the pattern matching may have multiple outcomes: a graph pattern *may*, *must* or *cannot* have a match on a partial model depending on whether the partial model can possibly be extended by adding model elements to fulfill the graph condition of the pattern.

Below, we introduce a novel graph rewriting technique for evaluating standard queries on incomplete, partial models. There are several challenges related to this matching problem. First, it has to cover an infinite number of possible instance models, which causes decidability issues [32], and necessitates the use of approximation techniques [33]. Also, the result set has to be calculated efficiently, as checking how a pattern may or may not match can cause a combinatorial explosion even for small partial models.

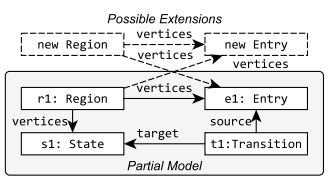
3.1 Partial Matches on Incomplete Models

When matching patterns on a complete model, the matches refer to real objects of the model in accordance with closed world semantics. However, when only a partial model is to be matched, a match may refer to currently non-existing objects, which are created in a later state of the development. Therefore, we rely upon projections of a full match of a complete model M to partial matches of partial model P where undefined elements refer to new elements added to M later.

A **projection of a match** on a partial model P is defined by a function π_P, which maps fully defined matches of an extended models M ($P \subseteq M$) to a partial match of P by excluding parameter bindings that are unavailable in P (but they remain defined for all parameters which maps an element of P to a parameter). Formally, $\pi_P(m) = m'$ if

$$m'(param) = \begin{cases} m(param) & \text{if } m(param) \in \mathsf{Objects}_P \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example For example, Figure 4a shows a partial model of a simple statechart with a single State, an Entry in a Region and a Transition. Some possible extensions are marked with a dashed line: either new Regions or new Entries can be added to the model, or they can be linked with vertices references.



(a) Example partial model with possible extensions.

```
pattern entryInRegion(r:Region, e:Entry) {
  Region.vertices(r,e);
}
```

Matches	r:Region	e:Entry
m_1	r1	e1
m_2	r1	new Entry
m_3	new Region	e1
m_4	new Region	new Entry

(b) Pattern *entryInRegion* and its possible matches.

Figure 4: Pattern matching for partial models

The graph pattern *entryInRegion* collects regions with their entry states. Figure 4b also enumerates the possible kind of matches:

m_1 is a complete match where all parameters are bound to an existing object of the partial model (e.g. r1 and e1) which satisfy the condition of the pattern.

m_2, m_3 are incomplete matches which represent a possible satisfaction of the pattern between some existing and some future objects. For example, m_2 represents a case where a new Entry needs to be added to r1 region with a new vertices reference to form a match.

m_4 is a symbolic match on a completely new partition of a model (where all of the elements are new), which represents all other matches that are independent from the partial model.

The use of symbolic values and match projection in the context of a partial model allows one to cover an infinite number of possible matches. In the following, we formalize when a pattern *must*, *may* or *cannot* be matched on a partial model.

3.2 Must and May Modality of Graph Queries

Now we will introduce two kinds of modalities to calculate possible matches of pattern p on partial model P. A *must-match* (denoted as $\square[P, m \models p]$) is a match which exists in every possible M extension of P (including P itself). Formally:

$$\square[P, m \models p] := \forall M : [P \subseteq M] \Rightarrow M, m \models p$$

A *may-match* (denoted as $\Diamond[P, m \models p]$) represents a case where a m is a projected match of an extension of P (even if the match does not exist directly in P). Formally:

$$\Diamond[P, m \models p] := \exists M : P \subseteq M \wedge M, m' \models p \wedge m = \pi_P(m')$$

Example Table 1 collects the must and may-matches of pattern `entryInRegion` shown in Figure 4b, when it is matched on the partial model of Figure 4a. In this example, the r1 and e1 pair is the only must-match: this match will exist in any possible extension of the partial model, i.e. it cannot be removed by adding new elements. The may-matches, as always, also contain the must-match, and additionally contain two cases with new elements: a new entry in r1 may create a new match, and a completely new region with an entry may also create matches. However, match m_3 requires the creation of an additional Region to the existing Entry is not a may-match, because the pattern requires a vertices containment reference and the creation of a second parent to e1, which would violate the containment hierarchy.

Table 1: Must and may-matches of an example model query on partial model

must	r:Region	e:Entry
m_1	r1	e1

missing	r:Region	e:Entry
m_4	new Region	e1

may	r:Region	e:Entry
m_1	r1	e1
m_2	r1	new Entry
m_3	new Region	new Entry

Combination of may- and must-matches Table 2 summarizes the possible combinations of may- and must-matches. If the partial model can be extended to a valid model (so $\exists M : P \subseteq M$), a must-match implies a may-match ($\square[P, m \models p] \Rightarrow \Diamond[P, m \models p]$), so a must and may-match simply means a must-match. If a match is a must-match, but not a may-match it means inconsistency (marked with an X). Because $P \subseteq P$ is true, if there is an actual match on the partial model, then, by definition, the must-matches and may-matches have to contain this match. Cases where this condition does not hold are marked with an X. A may-match but not must-match combination has the weakest consequence; namely a pattern might or might not be satisfied in an extension of the partial model. And finally, if an m is not a may-match it means that it cannot be matched in any possible extension of the model.

Table 2: Combination of concrete, must- and may-matches

	$\Diamond[P, m \models p]$		$\neg\Diamond[P, m \models p]$	
	$[P, m \models p]$	$\neg[P, m \models p]$	$[P, m \models p]$	$\neg[P, m \models p]$
$\square[P, m \models p]$	$\forall M : P \subseteq M$ $\Rightarrow [M, m \models p]$ (Constant match)	X	X	X
$\neg\square[P, m \models p]$	$[P, m \models p]$, $\exists M : P \subseteq M$ $\wedge\neg[M, m \models p]$ (Possibly disappearing match)	$\neg[P, m \models p]$, $\exists M : P \subseteq M$ $\wedge[M, m \models p]$ (Possibly appearing match)	X	$\forall M : P \subseteq M$ $\Rightarrow \neg[M, m \models p]$ (Impossible match)

Here, it is necessary to examine the semantics of must- and may-matches in the context of well-formedness constraints. If a pattern has a must-match of an ill-formedness pattern, then it marks an invalid model that cannot be repaired by adding new elements. If a partial model has a may-match ($\Diamond[P, m \models p]$), and the pattern has an actual match on the partial model ($[P, m \models p]$) it means that the model is currently ill-formed, but it might be repaired by adding new elements. Otherwise, if there is a may-match ($\Diamond[P, m \models p]$), but there no actual matches ($\neg[P, m \models p]$), it means that the model is currently well-formed, but the may-match highlights possible ways to inject errors.

4 Rewriting Model Queries with Open-World Semantics

Next, we introduce a novel technique that is able to produce must- and may-matches from existing model queries that are defined on complete models. The approach is based on a graph pattern rewriting technique that creates over- and under-approximated patterns to evaluate must-matches and may-matches.

4.1 Approximation of Must- and May-Matches

Unfortunately, to decide whether a provisional match of pattern p on a partial model P can be extended to a complete match for a model M where $P \subseteq M$ is a complicated problem. In general, it requires complex logic analysis [32], which scales poorly with respect to the size of the partial model [33], and may have undecidability issues. Therefore, we will introduce approximations to tackle this problem.

In our approach, must- and may-matches are approximated with regular pattern matching problems that can be efficiently evaluated on complete and partial models. A pattern matching problem $P^U, m^U \models p^U$ *under-approximates* a must-matching problem, if it satisfies the following constraint:

$$[P^U, m^U \models p^U] \quad \Rightarrow \quad \Box[P, m \models p]$$

Similarly, a pattern matching problem $P^O, m^O \models p^O$ *over-approximates* a may-matching problem, if the following constraint is satisfied:

$$\Diamond[P, m \models p] \quad \Rightarrow \quad [P^O, m^O \models p^O]$$

These formulae define a conservative approximation of must- and may-matches over partial models (see Table 3) with two potential inaccuracies: (1) a must-match might not be detected in a partial model, or (2) the unsatisfiability of a match might not be proved. In our approach, these inaccurate cases are collected in the same category as the may-matches. In other words, the match result is approximated in the direction of may-matches, which also collects the unknown cases. This is a safe compromise in most application areas like model validation.

Table 3: Consequences of approximated matches

	$[P^O, m^O \models p^O]$	$\neg[P^O, m^O \models p^O]$
$[P^U, m^U \models p^U]$	$\Box[P, m \models p]$	X
$\neg[P^U, m^U \models p^U]$	$\Diamond[P, m \models p]$ + Unknown	$\neg\Diamond[P, m \models p]$

In our approach, must- and may-matches are produced in four steps:

1. Approximated models P^U and P^O are created from the original partial model P (Section 4.2).

2. Approximated patterns p^U and p^O are constructed from p (Section 4.4) with the help of uncertain model indexers (Sections 4.3).

3. The approximated pattern matching problems are executed as normal pattern matching problems.

4. The matches of the approximated pattern matching problem m^U, m^O are interpreted as may- and must-matches of the original pattern p (Section 4.5).

In the following, we provide a constructive way for creating approximated models and patterns for a must/may matching problem, along with a technique for transforming over- and under-approximated matches back to must- and may-matches.

4.2 Representation of Partial Models

Now we will show how approximated models P^O and P^U can be created from a partial model P which will serve as a basis for evaluating approximated patterns over it later on. In our approach, standard Ecore [36] models are used to define partial models enriched with special annotations on model elements which are adapted from [14]. In our current implementation, both P^O and P^U are created in the same way from a PartialModel, which is illustrated in Figure 5.

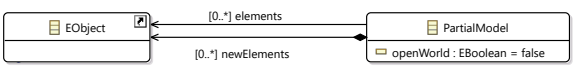


Figure 5: Helper metamodel for partial models.

This partial model representation has three important characteristics. These are:

- The reference elements represents the objects of the partial model. Our technique incorporates objects only if they are referred by the partial model with this reference. Hence it is easy to temporarily exclude elements from an existing model to create experimental variations of the model without modifying it.
- A partial model contains additional symbolic (newly created) elements along the newElements reference. A new element in a may-match represents a new object that needs to be created (in a proper context) in order to satisfy the condition of the match. Typically, a partial model contains a new element for each concrete (non-abstract) object of the metamodel. However, the analysis can be restricted by removing objects, thus forbidding the creation of this kind of object.
- The semantic interpretation of the partial model can be switched between open-world and close-world by setting the value of the attribute openWorld.

4.3 Open-World Indexing of Model Elements

The construction of the over- and under-approximated patterns is separated into two layers: (1) first an *uncertain property indexer* collects the possible variations

of model properties (like objects and references), and (2) an *uncertain pattern layer* that combines those uncertain properties into over- and under-approximated patterns. There we present the uncertain property indexer.

There are four kinds of basic properties of a partial model: (1) what the objects are (i.e. $\mathsf{Objects}_P$) (2) what the type of objects in the model is (C(o)) (3) where references are between the objects (R(s,t)), and (4) what the attribute values are in the model (A(o,v)). In the following, we describe how these basic properties can be over- and under-approximated by appropriate graph patterns.

Objects First, the objects of an extended model are either objects that are present in the partial model (referred by elements) or newly created objects (contained by newElements). However, for newly created elements referred in projected matches it is not decidable that the new objects are different or equal. Therefore the equivalence of the objects needs to be over- and under-approximated. Figure 6 illustrates a pattern for which under- and over-appriximating the equivalence of two objects, which are matched by `mustEqual` and `mayEqual` patterns. Objects e_1 and e_2 must be equals in all possible extensions of P^U, if they are equal in the partial model. However, two objects may be equal if they are represented by the same prototype object.

```
pattern mustEqual(p,e1,e2) {
  PartialModel.elements(p,e1);
  e1 == e2;
}
pattern mayEqual(p,e1,e2) {
  PartialModel.openWorld(p,true);
  PartialModel.newElements(p,e2);
  e1 == e2;
} or {
  find mustEqual(p,e1,e2);
}
```

Figure 6: Approximation of equivalences

```
pattern mustType_C(p,e) {
  C(e); PartialModel.elements(p,e);
}
pattern mayType_C(p,e) {
  C(e);
  PartialModel.newElements(p,e);
  PartialModel.openWorld(p,true);
} or {
  find mustType_C(p,e);
}
```

Figure 7: Approximation of type predicates

Types of objects It can be decided whether an object o must be of class C by simply evaluating the predicate $\mathsf{C}(o)$, which is a safe under-approximation. However, in the case of partial models, the set of instances of class C may include some elements from the symbolic instances with compatible types. Figure 7 illustrates the schema of patterns that match objects which are necessary or possible instances of class C.

References and attributes The indexing of possible and necessary references and attributes is a more complex case, as illustrated in the pattern template in Figure 8. First, in Figure 8, R(s,t) must be true if the reference is present in

the partial model, so it is a safe under-approximation. However, possible pairs of objects included in an over-approximation of R(s,t) have to satisfy several structural constraints (see Section 2.2 for more details):

1. **Type compliance:** If there is a reference between a source class S and a target class T, then a possible reference can only be instantiated between possible instances of S and T, so if the types are not correct, R(s,t) is excluded from the over-approximation.

2. **Upper multiplicity:** If there is an upper multiplicity defined for the reference (in the form of min..max), then the number of outgoing references must be less than max in order to possibly create a new reference. If this constraint fails, R(s,t) is excluded from the over-approximation.

3. **Upper multiplicity of inverse:** Similarly, if the reference S has an inverse reference I with an upper bound max, then the number of incoming R references to the target (which is the same as the number of outgoing I references from the target) must be less than max in order to possibly create a new reference.

4. **Containment reference, multiple parent:** There are two ways of violating the containment hierarchy with an additional reference. The first case is when an additional parent is created for an object. So, if there is a containment reference to a target object, then it is not possible to add another containment reference.

5. **Containment reference, circular containment:** Another way of violating the containment hierarchy is to create a circle with a new containment reference. Therefore it is not possible to create a containment reference between object s and t if there is a path of containments from t to s.

6. **Inverse of a containment reference:** If the inverse of R is a containment reference, then a may match has to satisfy **multiple parent** and **circular containment** rules.

In the following, we describe how the previous properties indexed with *may* and *must* modalities can be combined to create approximated patterns.

4.4 Transforming Approximated Patterns

A pattern p defines several structural constraints on a match m of model M; it is satisfied when then the pattern condition holds, which is denoted by $[M, m \models p]$. The condition of a pattern is defined as a disjunction of pattern bodies, which is defined by a set of constraints that has to be simultaneously satisfied:

$$[M, m \models p] \Leftrightarrow \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)$$

```
pattern mustReference_R(p, s, t) {
  PartialModel.elements(p, s);PartialModel.elements(p, t); S.R(s, t);
}
pattern mayReference_R(p, s, t) {
  // The partial model is Open World
  PartialModel.openWorld(p,true);
  // There are s and t with the correct types
  find mayType_S(p,s); find mayType_T(p,t);
  // Upper multiplicity of R allows the addition of a new reference
  numberOfExistingReferences == count find mustReference_R(p, s, _);
  check(numberOfExistingReferences < {upper multiplicity of R});
  // Upper multiplicity of the inverse reference I allows the addition
  numberOfOppositeReferences == count find mustReference_I(p, _, t);
  check(numberOfOppositeReferences < {upper multiplicity of I});
  // If R is a containment relation, the new reference cannot create
  //  1. Multiple parents
  neg find mustContains(p,_,t);
  //  2. Circle in the containment hierarchy
  neg find mustTransitiveContains(p,t,s);
  // If I is the inverse of R, and R is a containment relation,
  // then the new reference cannot create
  //  1. Multiple parents
  neg find mustContains(p,s,_);
  //  2. Circle in the containment hierarchy
  neg find mustTransitiveContains(p,s,t);
} or {
  find mustReference_R(p, s, t);
}

// Support patterns, where R_1 ... R_n are containment references
pattern mustContains(p,s,t)
  {mustReference_R_1(p,s,t);} or ... or {mustReference_R_n(p,s,t);}
pattern mustTransitiveContains(p,s,t) {
  find mustContains+(p,s,t);}
```

Figure 8: An approximation of reference predicates

Our approach is to create an over- and under-approximated version of the pattern condition by using the previously over- and under-approximated versions of atomic constraints. So an over-appriximated version of a pattern is created by over-approximating each constraint in it, which creates a valid overapproximation of the pattern:

$$\Diamond[P, m \models p] = \Diamond\Big[P, m \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)\Big] \Rightarrow$$

$$\Big[P^O, m^O \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} [constraint(m, var)]^O\Big] = [P^O, m^O \models p^O]$$

And similarly, an under-approximated version of a pattern is created when each constraint is replaced by an under-approximated one:

$$\Box[P, m \models p] = \Box[M, m \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)] \Leftarrow$$

```
1  // Original pattern
2  pattern noEntry(r : Region) {
3    neg find entryInRegion(r, _);
4  }
5
6  // Must version
7  pattern mustPattern_noEntry(p,r)
8  {
9    find mustType_Region(p,r);
10   neg find mayPattern_entryInRegion
11     (p,r,_);
12 }
13 // May version
14 pattern mayPattern_noEntry(p,r)
15 {
16   find mayType_Region(p,r);
17   neg find mustPattern_entryInRegion
18     (p,r,_);
19 }
```

```
20 // Original pattern
21 pattern multipleEntry(r) {
22   find entryInRegion(r, e1);
23   find entryInRegion(r, e2);
24   e1 != e2; }
25 // Must version
26 pattern mustPattern_multipleEntry(p,r){
27   find mustPattern_entryInRegion(
28     p,r,e1);
29   find mustPattern_entryInRegion(
30     p,r,e2);
31   neg find mayEqual(p,e1,e2); }
32 // May version
33 pattern mayPattern_multipleEntry(p,r){
34   find mayPattern_entryInRegion(
35     p,r,e1);
36   find mayPattern_entryInRegion(
37     p,r,e2);
38   neg find mustEqual(p,e1,e2); }
```

Figure 9: Examle under- and overapproximated patterns

$$[P^U, m^U \models p^U] = [M^U, m^U \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} [constraint(m, var)]^U],$$

where $[constraint]^U$ symbolizes the under-approximated (must) version of constraint, and $[constraint]^O$ denotes the over-approximated (may) version of it. In the latter modality, the variables and parameters can be bound to symbolic objects, which is handled by our open world indexing implementation by matching newElements prototypes. The only remaining task is to replace constraints of the original match by calling the must or may variant of the corresponding constraint. In Figure 9, there are two patterns with under- and over-approximated patterns:

- **Pattern:** Newly created patterns are prefixed with `mustPattern_` or `mayPattern_` and add a parameter `p` for the PartialModel object. For instance, as shown in Figure 9, `mustPattern_noEntry(p,r)` is the under-approximated version of `noEntry(r)`.
- **Classifier Constraint:** We replace all C`(e)` constraints by new pattern calls to `find mustType_`C`(p,e)` or `find mayType_`C`(p,e)`.
- **Path Constraint:** We split path expressions into single reference and attribute constraints, and replace all occurrences of a single R`(s,t)` constraint by either `find mustReference_`R`(p,s,t)` or `find mayReference_`R`(p,s,t)`.
- **Equality Constraints:** in the case of equality constraints, replace all `a==b` by `find mustEqual(p,a,b)` or `find mayEqual(p,a,b)`. However, in the case of inequality (`a!=b`), the modality has to be changed to the dual:
  - [`a!=b`]^U is replaced by `neg find mayEqual(p,a,b)`
  - [`a!=b`]^O is replaced by `neg find mustEqual(p,a,b)`

For example, the constraint `e1!=e2` in line 24 of Figure 9 expresses the fact, that the two entries are different. In the must version of the pattern, it is replaced by `neg find mayEqual(p,e1,e2)`, which excludes matches where the two entries can be mapped to the same element. However, in the may version of the pattern, it is replaced by `neg find mustEqual(p,e1,e2)`, which exclude matches only where it is certain that the two matches are equal.

- **Pattern Call Constraints:** There are different rules to map pattern calls:
  - **Positive call:** a positively called pattern `find` *ref* `(par)` is mapped to `find mustPattern_`*ref*`(p,par)` or `find mayPattern_`*ref*`(p,par)`. For instance, `find entryInRegion(r, e1);` in line 22 of Figure 9 is replaced by `find mustPattern_entryInRegion(p,r,e1)` in the must variant of the pattern.
  - **Negative call:** In the case of a negative pattern call (`neg find`), the modality of the called pattern has to be changed to the opposite:
    * [`neg find` *ref* `(par)`]^U replaced by `neg find mayPattern_`*ref*`(p,par)`
    * [`neg find` *ref* `(par)`]^O replaced by `neg find mustPattern_`*ref*`(p,par)`

    For example, the negative pattern call `neg find entryInRegion(r,_)` in line 3 of Figure 9 is transformed to a negative may-pattern call `neg find mayPattern_entryInRegion(p,r,_)` constraint in the must version of the pattern, which forbids all possible matches where the entry can be in the checked region. Conversely, in the may version, it is transformed to `neg find mustPattern_entryInRegion(p,r,_)`, which filters the matches only where it is certain that there is an entry in the region.
  - **Transitive closure:** A transitive closure call `find` *ref*`+(par)` is transformed to either an under-approximated `find mustPattern_`*ref*`+(p,par)` or an over-approximated `find mayPattern_`*ref*`+(p,par)`.
- **Count find:** The number of occurrences of a pattern may be under- and over-approximated in several cases by replacing `C==count find` *ref* `(par)` by `C<=count find mustPattern_`*ref*`(par)` or `C>=count find mayPattern_`*ref*`(par)`. However, as the count of the matches (i.e. the value of `C`) can be used in other constraints, this constraint is supported only when `C` is a constant integer.
- **Eval, check:** In those features language that cannot be supported automatically, under- and over-approximated versions of the embedded expressions need to be created manually.

4.5 Interpretation of approximated matches

Calculating approximated matches The over- and under-approximated versions of matches can be matched on partial models (as regular instance models)

by traditional graph pattern matching approaches. For this purpose, we have been using the incremental graph pattern matching engine of the VIATRA framework [4]. Since the approximated patterns are more complex than the original pattern, we expect that matching partial models will be slower. According to our initial experiments, we expect a quadratic decrease in performance - which is still more efficient than using SAT/SMT solvers for the same purpose.

Interpreting approximated matches The calculated matches of approximated patterns on partial models need to be interpreted as must- and may-matches by projecting them onto the original partial model ($m = \pi_P(m^O)$ and $m = \pi_P(m^U)$). This projection is not computationally intensive since matches are only filtered but never extended.

4.6 Validation of the approach

In order to validate our approach, we developed automated transformations to (1) rewrite models into partial models, and (2) rewrite graph patterns into approximated patterns. We executed these transformations in the context of Yakindu statecharts (i.e. the running example of our paper).

Evaluating must and may matches on a partial model is always decidable, and the problem has polynomial time complexity (as it can be reduced to a subgraph isomorphism problem [27]). In contrast, logic analysis of graph patterns and partial models is, in general, undecidable [32]. Also, industrial graph query engines like [4] are able to efficiently evaluate graph queries over large models with thousands of objects. In our example, the execution time of this compilation step (i.e. the rewrite and execution of transformations) was negligible. Still, our previous experience showed that advanced SAT and SMT solvers failed to solve partial model analysis with the same metamodel and size [33].

Besides the abov, we systematically developed a test set with full metamodel coverage [6] and full inclusion of well-formedness constraints defined for Yakindu statecharts, which consists of prototype graph patterns. This test set is listed in the Appendix, and the generated output and further details are available in GitHub[1]. We generated both a may- and a must-approximation from the graph pattern of the designated constraint, and we evaluated these patterns on sample prototypical instance models. The correctness of the generation was established by manually inspecting the retrieved result set of the may and must patterns. In the future, we intend to carry out a more systematic performance evaluation of our approach.

[1] `https://github.com/FTSRG/publication-pages/wiki/Evaluating-Well-Formedness-Constraints-on-Partial-Models`

5 Related Work

5.1 Analysis of Uncertain Models

Partial models are a subclass of *uncertain models*, which offer a rich specification language [14, 28] amenable to analysis. Uncertain models provide a more expressive language compared to partial models, but without handling additional WF constraints. Such models document semantic variation points generically via by means of annotations on a regular instance model. Most uncertain model analysis approaches focus on the generation of possible concrete models or the refinement of partial models. A potential concrete models compliant with an uncertain model may be synthesized by the Alloy Analyzer and its back-end SAT solvers [31, 30], or be refined by graph transformation rules [29].

The most similar approaches [15, 16] analyze the possibility of model transformation rule matching and execution on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"). The main difference is that in their approach inspects possible partitions of a concrete model (so checking all possible P-s for a given M with $P \subseteq M$), instead of possible extensions of a partial model. Therefore, it cannot be used to support a mostly incremental development process with growing models.

There is an extensive tool support for editing and analyzing advanced uncertain models [5]. In contrast, our approach may be applied on (unfinished) EMF models directly [4] in their own editors, to indicate must- and may-matches of ill-formedness constraints as errors and warnings.

5.2 Verification of Model Transformations

Besides uncertain models, there are several formal methods available that seek to evaluate graph patterns on abstract graph models (either abstract interpretation[25] or predicate abstraction [26]) in order to detect possible concretizations matches. Those techniques typically employ similar techniques called pre-matching to create may-matches that are further analyzed.

5.3 Logic Solver Approaches

There are several approaches available that map partial models and WF constraints into a logic problem, which are solved by underlying SAT/SMT-solvers. With these techniques the implication between a partial model and the satisfaction of a well-formedness constraint can be directly evaluated in order to reason about must- and may-matches.

Complete frameworks with standalone specification languages include Formula [20] (which uses the Z3 SMT- solver [13]), Alloy [19] (which relies on SAT solvers like Sat4j[22]) and Clafer [2] (using backend reasoners like Alloy).

There are several approaches which seeks to validate standardized engineering models enriched with OCL constraints [17] by relying upon different back-end

logic-based approaches such as constraint logic programming [11, 10, 8], SAT-based model finders (like Alloy) [34, 1, 9, 21, 35], first-order logic [3], constructive query containment [24], higher-order logic [7, 18], and rewriting logics [12]. Partial snapshots and WF constraints can be uniformly represented as constraints [32].

The scalability of all these approaches are limited to small models / counterexamples. Furthermore, these approaches are either a priori bounded (where the search space needs to be explicitly restricted) or they have decidability issues [33].

6 Conclusions and Future Work

Here we presented a novel validation technique that is able to check unfinished, partial models with well-formedness constraints that are defined for fully defined models. The outcome of searching a malformed model partitions can be either a must-match (which detects whether a partial model already contains a conceptual flaw), a may-match (which means that a possible extension may become invalid, and highlights all possible threats), or no match (which proves that the partial model already satisfies a validation rule). The approach is based on an approximation technique that reduces must- and may-matching to regular pattern matching problems, which can be executed by graph query engines.

As a future work we are planning to integrate our pattern matching technique with advanced partial modeling formalisms like [14] to increase the expression power of the analysis. Also, we intend to carry out a systematic performance evaluation, and experiment with guiding incremental model generators processes [33] by evaluating must- and may-matches on intermediate solutions.

References

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[2] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, pages 1–35, 2013.

[3] B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proc. of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[4] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In *Fourth International Conference on Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 167–182. Springer, 2011.

[5] Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors. *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. IEEE Computer Society, 2015.

[6] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06.*, pages 85–94, Nov 2006.

[7] A. D. Brucker and B. Wolff. The HOL-OCL tool, 2007. `http://www.brucker.ch/`.

[8] Fabian Büttner and Jordi Cabot. Lightweight string reasoning for OCL. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349 of *LNCS*, pages 244–258. Springer, 2012.

[9] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL transformations using transformation models and model finders. In *14th International Conf. on Formal Engineering Methods,ICFEM'12*, pages 198–213. LNCS 7635, Springer, 2012.

[10] J. Cabot, R. Clariso, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conf. on*, pages 73–80, April 2008.

[11] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 547–548, New York, NY, USA, 2007. ACM.

[12] M. Clavel and M. Egea. The ITP/OCL tool, 2008. `http://maude.sip.ucm.es/itp/ocl/`.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering*, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.

[15] Michalis Famelis, Rick Salay, and Marsha Chechik. The semantics of partial model transformations. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 64–69. IEEE Press, 2012.

[16] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. Transformation of models containing uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*, pages 673–689. Springer, 2013.

[17] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4:386–398, 2005.

[18] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 152–166. Springer, 2009.

[19] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[20] Ethan K Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Model Driven Engineering Languages and Systems*, pages 653–667. Springer, 2011.

[21] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into use. In *TOOLS'11 - Objects, Models, Components and Patterns*, volume 6705 of *LNCS*, pages 290–306, 2011.

[22] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[23] The Object Management Group. *Object Constraint Language, v2.0*, May 2006.

[24] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, 73:1–22, 2012.

[25] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157(1):39–59, 2006.

[26] Thomas W Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *International Conference on Computer Aided Verification*, pages 15–30. Springer, 2004.

[27] Grzegorz Rozenberg. *Handbook of Graph Grammars and Comp.*, volume 1. World scientific, 1997.

[28] Rick Salay and Marsha Chechik. A generalized formal framework for partial modeling. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *LNCS*, pages 133–148. Springer Berlin Heidelberg, 2015.

[29] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. A methodology for verifying refinements of partial models. *Journal of Object Technology*, 14(3):3:1–31, 2015.

[30] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 938–945. IEEE, 2012.

[31] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 224–239. Springer Berlin Heidelberg, 2012.

[32] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and Systems Modeling*, pages 1–36, 2015.

[33] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 87–103, 2016.

[34] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.

[35] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation and Test in Europe, (DATE'10)*, pages 1341–1344. IEEE, 2010.

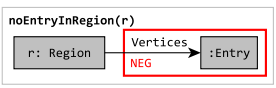
[36] The Eclipse Project. *Eclipse Modeling Framework*. `//www.eclipse.org/emf`.

[37] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, October 2007.

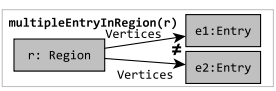
[38] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[39] Yakindu Statechart Tools. *Yakindu*. `http://statecharts.org/`.

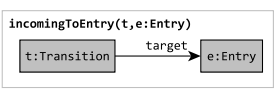
Appendix



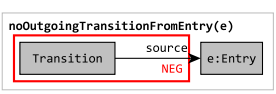

```
pattern entryInRegion(r:Region, e:Entry) {
  Region.vertices(r,e); }
@Constraint pattern noEntryInRegion(r:Region) {
  neg find entryInRegion(r,_); }
```



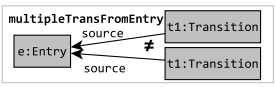

```
@Constraint pattern multipleEntryInRegion(r) {
  find entryInRegion(r,e1); find entryInRegion(r,e2);
  e1 != e2; }
```



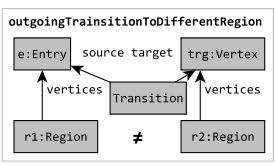

```
pattern transition(t,src,trg) {
  Transition.source(t,src); Transition.target(t,trg); }
@Constraint pattern incomingToEntry(t, e:Entry) {
  find transition(t,_,e); }
```



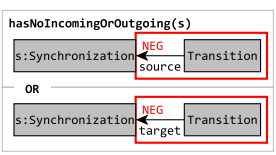

```
@Constraint
pattern noOutgoingTransitionFromEntry(e:Entry) {
  neg find transition(_,e,_);
}
```



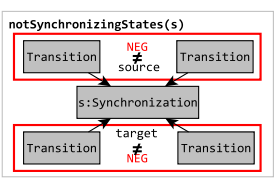

```
@Constraint pattern multipleTransitionFromEntry(e:Entry,t1,t2) {
  find transition(t1,e,_); find transition(t2,e,_);
  t1! == t2;
}
```



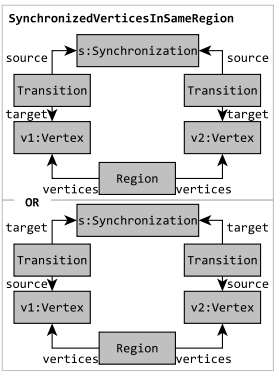

```
@Constraint
pattern outgoingTrainsitionToDifferentRegion(e:Entry,trg,r) {
  find transition(_,e,trg);
  Region.vertices(r1,e);
  Region.vertices(r2,trg);
  r1 != r2;
}
```



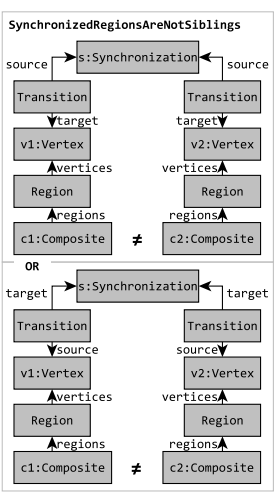

```
@Constraint hasNoIncomingOrOutgoing(s:Synchronization) {
  neg find transition(_,_,s);
} or {
  neg find transition(_,s,_);
}
```




```
private pattern hasMultipleOutgoingTrainsition(v) {
  find transition(_,v,trg1); find transition(_,v,trg2); trg1 != trg2; }
private pattern hasMultipleIncomingTrainsition(v) {
  find transition(_,src1,v); find transition(_,src2,v); src1 != src2; }
@Constraint notSynchronizingStates(s:Synchronization) {
  neg find hasMultipleOutgoingTrainsition(s);
  neg find hasMultipleIncomingTrainsition(s); }
```




```
@Constraint
pattern SynchronizedVerticesInSameRegion(s:Synchronization,v1,v2) {
  find transition(t,v1,s);
  find transition(t,v2,s);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
} or {
  find transition(t,s,v1);
  find transition(t,s,v2);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
}
```



```
@Constraint
pattern SynchronizedRegionsAreNotSiblings(s:Synchronization,v1,v2) {
  find transition(t,v1,s);
  find transition(t,v2,s);
  CompositeElement.regions.vertices(r1,v1);
  CompositeElement.regions.vertices(r2,v2);
  r1 != r2;
} or {
  find transition(t,s,v1);
  find transition(t,s,v2);
  CompositeElement.regions.vertices(r1,v1);
  CompositeElement.regions.vertices(r2,v2);
  r1 != r2;
}
```

Acta Cybernetica **23** (2017) 715–735.

Automating the Refactoring Process

Gábor Szőke[a]

Abstract

To decrease software maintenance cost, software development companies use static source code analysis techniques. Static analysis tools are capable of finding potential bugs, anti-patterns, coding rule violations, and they can also enforce coding style standards. Although there are several available static analyzers to choose from, they only support issue detection. The elimination of the issues is still performed manually by developers.

Here, we propose a process that supports the automatic elimination of coding issues in Java. We introduce a tool that uses a third-party static analyzer as input and enables developers to automatically fix the detected issues for them. Our tool uses a special technique, called reverse AST-search, to locate source code elements in a syntax tree, just based on location information. Our tool was evaluated and tested in a two-year project with six software development companies where thousands of code smells were identified and fixed in five systems that have altogether over five million lines of code.

Keywords: refactoring, code smells, reverse ast-search, spatial index

1 Introduction

Refactoring is a common practice for improving software maintainability. By definition, refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [6]. The original declaration introduces the understandability and changeability aspect of refactoring. However, other research shows that the idea can also be applied to other purposes [18], such as improving performance, security, and reliability. Developers use it regularly to transform code into a more maintainable form. About 70–80% of all structural changes in the code are due to refactorings [32, 2]. Industrial case studies [14] revealed that the refactoring definition in practice is not confined to a rigorous definition of semantics-preserving code transformations and that developers perceive that refactoring involves substantial cost and risks.

Many IDEs (integrated development environments) offer refactoring features to provide assistance in these regular tasks. Eclipse, for instance, has a separate *Refactor menu* where such operations are available, e.g., developers can rename a source

[a]University of Szeged, E-mail: gabor.szoke@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.16

code element (e.g., a variable) and the IDE will correct all its references. Similarly, it is possible to extract local variables, methods, classes, or move elements, among several other operations for refactoring purposes.

Tools which support automatic refactorings often assume that programmers already know how to refactor, and they have knowledge about the catalog of refactorings [6], but this is typically not a reasonable assumption. As Pinto et al. found in their study where they examined questions of refactoring tools on Stack Overflow, programmers are usually unable to identify refactoring opportunities, because of lack of knowledge in refactoring, or do not understand of the legacy code. They also claim that "*refactoring recommendations is one of the features that most of Stack Overflow users desire (13% of them)*" [24].

In order, to offer refactoring recommendations to developers, some studies [5] use static code analyzers to report problematic code segments. Static analysis tools are capable of finding potential bugs, anti-patterns, and coding rule violations. Fontana et al. compare the capabilities of refactoring tools to remove code smells, and they identify only one tool (*JDeodorant*) which can support code smell detection and then to suggest which refactoring to apply to remove the detected smells.

In this study, we introduce a process which suggests refactoring opportunities for coding issues and it is able to automatically refactor them. Our solution uses the PMD static analyzer to find Java coding rule violations. We created several refactoring transformations that support fixing these coding issues.

To perform refactoring operations, we created a mapping between the textual output of the static analyzer and the structural representation of the source code. This task required us to create an algorithm that takes textual source code position (i.e. start and end line) and type information (i.e. `for` loops) of the problematic code segment and executes a search on the syntax tree to locate the related source code element in the tree. To make reverse searching possible, we use a *spatial database*. The database is created by transforming the source code into a *geometric space*. Line numbers and column positions from the AST are used to define areas. These areas are used in *R-tree*s, where area-based searching is possible. To evaluate our approach we created a tool that applies the proposed process. The evaluation showed that our approach is adaptable in a real-life scenario, and it can provide viable results. Our tool was used in a project where it assisted in more than 6,000 automated refactorings covering systems ranging from 200 to 2,500 kLOC.

2 Overview

This study was part of an R&D project supported by the EU and the Hungarian Government. The goal of the 2-year project was to develop a software refactoring framework, methodology and software tools to support the "continuous reengineering" methodology, hence provide support to identify problematic code parts in systems and to refactor them so as to enhance maintainability. During the project, we developed a semi-automatic refactoring framework [29] and tested it on the source code of the industrial partners, having an *in vivo* environment and live feedback on

the tools. This is why partners not only participated in this project to help develop the refactoring tools, but they also tested and used the toolset on the source code of their own products. This provided a good chance for them to refactor their code and improve its maintainability.

During the development of the framework, we found several interesting problems and solutions to these problems which we think worth sharing.

Before going into details, we should overview the general refactoring process. As Fowler says, refactoring is *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [6]. Based on this definition if we model the code as a graph – which every compiler does – a refactoring is a (behavior preserving) transformation on a graph. More specifically, it can be viewed as a transformation on the *abstract syntax tree (AST)*. Executing such transformation requires three components. They are:

- An AST as a representation of the source code.
- A transformation algorithm.
- Starting points (AST nodes) called the *„origin"* where the transformation algorithm begins.

First, we parse the source code with a parser which builds an AST. Second, we create a transformation algorithm that will make modifications on the AST (i.e. pull up a method from one class to its ancestor). Next, we pick a node on the AST as the origin where the transformation algorithm will start working (i.e. selecting the method to pull up). After the transformation is performed, we get a modified (refactored) AST and the refactoring is complete.

Because our goal was to create automated refactorings, we extended the process with a few additional steps. These steps allow us to interact with developers and to make the transformations automatically. Our process works as follows:

Process 1.

1. *We create an AST representation of the source code.*
2. *We use static analysis on the source code to find problematic code parts, i.e. coding issues, rule violations, metric warnings. We list these issues as suggestions to help the user in finding candidates for refactoring.*
3. *The user selects one of the issues as the target of the refactoring.*
4. *Based on the type of the issue chosen by the user, a refactoring algorithm is selected that is capable of fixing the given type of issue.*
5. *Based on the selected issue a proper origin node is chosen from the AST.*
6. *The algorithm makes the transformation and modifies the AST, while keeping track of what modifications it made. A modified AST is created.*
7. *We generate source code from the modified AST.*
8. *The newly refactored source code is shown to the user where he or she can test the code and decide whether to accept or reject the refactoring.*

Next, we will present these steps in greater detail and discuss our findings.

3 Process Details

3.1 Building AST

To build an AST from the source code we use the SourceMeter [7] tool. SourceMeter uses OpenJDK [21] as a backend to parse the code and build an *abstract semantic graph (ASG)*. ASG is an extended version of the AST with cross-edges and much more [4]. This extra information was crucial in the automation process. Hence it allowed us to create flawless transformations which otherwise wouldn't be possible. *Thus Step 1 of Process 1 is covered.*

3.2 Finding Refactoring Suggestions

Choosing which part of the source code to refactor is quite hard. To improve the maintainability of the code, one can either start optimizing for metric values or try to get rid of anti-patterns by introducing design patterns to the code. Any of them might be a good solution. However, we will choose coding issues as the main target of our automated refactorings because our preliminary study [27] suggested that this is what developers want the most. To identify coding issues we choose the well known PMD static source code analyzer. It is a widely used and accepted tool among developers, especially for checking Java rule violations. Because all of the participating project members had a Java code base, it was an optimal choice to integrate it into the framework.

The output of PMD worked well in identifying coding issues and even in presenting some of these to developers as refactoring suggestions. Let us examine the sample in Listing 1. In this simple example, PMD finds 9 rule violations (with default settings). It finds issues like missing package declaration, missing comments, short variable names, magic numbers, missing braces and the list goes on. Even in this simple sample of code, there are many issues that can be fixed with computer assistance.

To extract the issues we use the XML output of PMD. This file contains a list of violations for each file with name, description, priority, and position information. An example violation is shown in Listing 2. This is one taken from the list of issues we got as output after running PMD on Listing 1. It clearly states that we should use curly braces in the `if` statement in Line 3.

After presenting these kinds of issues to developers, they are usually able to select one for refactoring. *Thus Step 2 and 3 of Process 1 are covered.*

3.3 Selecting the *Right* Transformation for the Job

After the user selected an issue, the next step is to find the right transformation. Many researchers are working on solutions to automate this process using machine learning techniques [11, 19, 22]. However, we used a much simpler approach. We created several general refactoring transformations, i.e. for moving, adding, deleting, and swapping source code elements. We created a mapping between PMD

Listing 1: Example code

```
1  public class Example {
2      public static int limiter(int x) {
3          if (x > 10)
4              return 10;
5          return x;
6      }
7  }
```

Listing 2: PMD's XML report

```
<file name="Example.java">
    ...
    <violation beginline="3" endline="4" begincolumn="3" endcolumn="
        13" rule="IfStmtsMustUseBraces" ruleset="Braces" class="
        Example" method="limiter" priority="3">
    Avoid using if statements without curly braces
    </violation>
    ...
</file>
```

violations and the transformations. For example, to fix the curly braces issue, we mapped it to an insertion transformation where a *block* element will be injected below the `if` statement (See the illustration in Figure 1). Each mapping defines different parameters based on the type of issue. The transformation in the former example requires an `if` statement as a parameter. *Thus Step 4 of Process 1 is covered.*

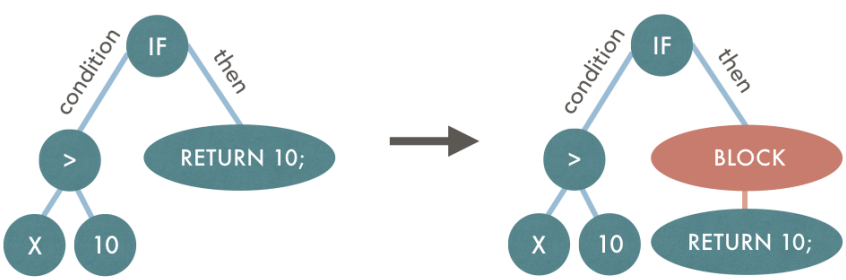


Figure 1: Simplified illustration of a refactoring on the AST of Listing 1.

Listing 3: PMD highlight

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

Listing 4: SourceMeter highlight

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10)
4             return 10;
5         return x;
6     }
7 }
```

3.4 Selecting a Proper Origin

After the user has selected an issue to fix and we chose the right transformation, the next step in the process is to choose an origin point on the AST, which we can give as a parameter to the transformation algorithm. In other words, one has to perform a search on the AST to find an element that matches both the description provided by the PMD report and the type of the parameter the transformation algorithm requires.

The report provides only a few key point for a violation, i.e. beginning line and column, ending line and column, class, and method. Also, the source file is available in the `file` tag. If we look at the example in Listing 2 and the results on the example in Listing 1, we get the problematic code part *highlighted*. The highlighted part in Listing 3 shows the particular `if` statement that needs braces. Although this highlighted segment is a good visual aid for the developer to find where the violation is, it is problematic for the computer to find nodes in the AST based on little more than position information.

A way to address the problem is to store position information for each source code element in the AST during the parsing process. Fortunately, SourceMeter does this already. Now that we know the positions on the AST, we can attempt to match the violation location information to the ones we have on the AST. It may come as no surprise that a simple equality match did not work. PMD and SourceMeter have different parsers and therefore it is more unlikely they will have the same position information for each and every source code element. Listing 4

shows the position SourceMeter has identified for the `if` statement in question. Looking at both highlighted cases reveals that a simple approximation will not suffice to get a match. To handle the problem, we took a different direction which we call *reverse AST-search*.

Reverse AST-Search

This idea was born when we decided to take a different direction and start looking at the source code elements, not as just data or nodes in a tree. In the text editor, they look like as little areas or patches. Since they all have begin and end lines and columns, they can be viewed as coordinates on a map. This led us to the idea of transforming the source code into a *geometric space*.

We took line numbers and column positions from the AST and used them to define areas. These areas form rectangles where the corner points are the begin and end positions of the source elements. The rectangular areas are then used to build a *spatial database*, where area-based queries are possible.

Spatial Databases and R-trees A spatial database [25] is a database that is optimized to store and query data that represents objects defined in a geometric space. Common database systems use indexes to quickly look up values and the way that most databases index data is not optimal for spatial queries. Instead, spatial databases use a *spatial index* to speed up database operations. To create spatial index data, we decided to use *R-trees* [10].

An R-tree is a data structure where the key idea is to group together information based on spatial data and index these groups by using their minimum bounding rectangles[1]. Next, these groups are bound together at the next level of the tree by their minimum bounding rectangles, and so on. This way, a query cannot intersect any of the objects contained because all the objects within a bounding rectangle occur together. The input of a search is a rectangle called a *query box*. Every rectangle in a node (starting from the root node) is checked to see whether it overlaps with the search rectangle or not. If it does, the same thing happens with its corresponding child nodes. The search goes on recursively until all matching nodes get visited. Meanwhile, when a leaf node is found and it overlaps with the query box it is added to the result set.

R-tree applications cover a wide spectrum, ranging from spatial and temporal to image and video databases. In industry, it is used where multi-dimensional data needs to be indexed. For example, a common application is in digital maps where R-trees are used to link geographical coordinates to POIs [15].

Building Spatial Index for the AST To create a spatial database for the source code, we used the *si* method in Algorithm 1. The method requires C, an AST element with position information. This might be a root node or a class node, say. When the algorithm commences, it creates an R-tree for storing the spatial

[1]The "R" in R-tree is for rectangle.

index *(Alg. 1 Line 1)* and begins to traverse the descendants of the C element *(Alg. 1 Line 2)*. Note that every kind of traversal is acceptable since the position of the elements do not depend on each other.

Algorithm 1 Building Spatial Index for the AST.

Funct $si(C)$
Require: C is an AST element
1: Let $\mathcal{I}$ be a new R-Tree
2: **for all** $c \in descendants(C)$ **do**
3: Let $P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}$ be the position coordinates of c
4: **if** $P_{start_{line}} = P_{end_{line}}$ **then**
5: Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
6: **else**
7: Add rectangle $\{P_{start_{line}}, P_{start_{col}}, P_{start_{line}}, \infty\}$ to $\mathcal{I}(c)$
8: Add rectangle $\{P_{end_{line}}, 0, P_{end_{line}}, P_{end_{col}}\}$ to $\mathcal{I}(c)$
9: **if** $P_{end_{line}} - P_{start_{line}} > 1$ **then**
10: Add rectangle $\{P_{start_{line}} + 1, 0, P_{end_{line}} - 1, \infty\}$ to $\mathcal{I}(c)$
11: **end if**
12: **end if**
13: **end for**
14: **return** $\mathcal{I}$

For each source code element c, the algorithm takes into account their position P; namely, start line, start column, end line, end column *(Alg. 1 Line 3)*. Next, using these positions rectangles are created. To be precise, it creates one, two or three rectangles, depending on the length of the current source element. If the element position is confined within a single line, one rectangle is created *(Alg. 1 Line 5)*, which is a line on the 2D plane. In the case of multiple lines, we have a multiline element, which is an element that starts at a line in a column, and it ends in another line in a column. All positions between these two positions are part of the element. For example, in Listing 4 the `if` statement is a two-line element and the highlight indicates the positions belonging to the statement. To handle this, we create two rectangles. The first line has no end column *(Alg. 1 Line 7)*, and the second begins at zero *(Alg. 1 Line 8)*. If there more lines between them, we create a rectangle that covers all the space between the two lines *(Alg. 1 Line 10)*.

Each time a rectangle is created it is added to the R-Tree with a binding to the AST element. This way when the spatial query function starts running, we will get AST elements instead of rectangles. Once all the descendant source code elements get visited, we can return the resulting R-tree, and the spatial index is ready.

When we tested the algorithm in the example in Listing 1, we got the search space shown in Listing 5. Note that every node that has multiple lines are separated into more rectangles, such as the `if` statement.

Listing 5: Search space for the example in Listing 1 with the rectangles and the type of their referred source code element

```
Rect (1, 1), (1, INF) = Class
Rect (2, 0), (6, INF) = Class
Rect (7, 0), (7, 2) = Class
Rect (2, 2), (2, INF) = Method
Rect (3, 0), (5, INF) = Method
Rect (6, 0), (6, 3) = Method
Rect (2, 16), (2, 19) = PrimitiveTypeExpression
Rect (2, 28), (2, 33) = Parameter
Rect (2, 28), (2, 31) = PrimitiveTypeExpression
Rect (2, 35), (2, INF) = Block
Rect (3, 0), (5, INF) = Block
Rect (6, 0), (6, 3) = Block
Rect (3, 3), (3, INF) = If
Rect (4, 0), (4, 14) = If
Rect (3, 6), (3, 14) = ParenthesizedExpression
Rect (3, 7), (3, 13) = InfixExpression
Rect (3, 7), (3, 8) = Identifier
Rect (3, 11), (3, 13) = IntegerLiteral
Rect (4, 4), (4, 14) = Return
Rect (4, 11), (4, 13) = IntegerLiteral
Rect (5, 3), (5, 12) = Return
Rect (5, 10), (5, 11) = Identifier
```

The Search Algorithm Once we have built our spatial index, we can use it to locate a node in the AST based on position information. We created a method that uses inputs such as the ASG from SourceMeter, the parameter type of the transformation, and the violation position from the PMD report in order to search the geometric space. The **R**everse **AST**-search **A**lgorithm (*rasta* for short) is listed in Algorithm 2 below.

Algorithm 2 The reverse AST-search algorithm.

Funct rasta(C, P, t)
Require: C is an AST element
Require: P is a position
Require: t is a type
1: Set the candidate list $\mathcal{R} := \{\}$
2: Compute $si(C)$
3: Let S be the set of all AST nodes whose have intersecting rectangles with P
4: **for all** $s \in \mathcal{S}$ **do**
5: **if** $type(s)$ is t **then**
6: Store s in $\mathcal{R}$
7: **end if**
8: **end for**
9: **return** $\mathcal{R}$

The purpose of the algorithm is to find the source code element that is highlighted in PMD's report. The function begins by creating a list of the source code element candidates. Next, it builds a spatial index with the C AST parameter *(Alg. 2 Line 2)*. The newly constructed index is used in the next step to query the candidates. As mentioned earlier, the spatial database requires a rectangle, called the query box as a search parameter. The query box in our case is the "highlight" from the PMD's output. We use this box to ask the R-tree which previously added rectangles intersect with the parameter. The R-tree returns with a set of AST nodes whose rectangles satisfied the query *(Alg. 2 Line 3)*. As an example, Listing 5 shows which source code elements (highlighted lines) remain after the query has been performed with the `Rect(3,3)(4,13)` box as the parameter.

Even with a small sample like Listing 5, the resulting set of the query can be quite big. To narrow the result set we use the third parameter, namely the type of the input parameter of the refactoring transformation, to filter the results *(Alg. 2 Line 5)*. The filtering is achieved by going through the result set and by inserting only those source code elements onto the candidate list whose type matches (actually, whose type is compatible with) the input type. After the filtering, the candidate list is returned as the result of the function.

To continue with the example in Listing 5, we have to filter the result set with the input parameter type of the refactoring transformation. In Section 3.3 we identified this source code element type as an `if` statement. Even with a quick glance, we can see that there are two rectangles where their type is an `if` statement. Since both

of the two rectangles refer to the same `if` element, the algorithm terminates. We have found an origin where the refactoring transformation can begin its operation.

Heuristics As we saw previously, the best case scenario is when the algorithm ends up having only a single element in the list of candidates. This happens when the result set has only one source code element with the type of parameter. In this case, it is evident which element was highlighted as the source of the violation. Nevertheless, there are times when the candidate list has multiple source code elements of the same type. In such cases, we have to select the proper element as the origin, otherwise the refactoring will be executed wrongly.

To remove ambiguity we decided to use a "distance" measure to find the best candidate. We defined the distance as the number of characters between the position of the source code element and the highlight. More specifically, on one hand, it is a metric of the number of characters between the start position (line and column) of the source code element and the start position (line and column) of the highlight. On the other hand, it is the number of characters between the end position (line and column) of the element and the end position (line and column) of the highlight. The distance is just the sum of the two. To calculate this value we use the original source file where the exact number of characters could be measured.

This method allows us to select the proper source code element from the list of candidates in almost every case. Still, there is a certain mathematical probability that the calculated distances will be equal for all candidates. However, because the chance of this event is astronomically small (it never even happened once in our exhaustive testing period, see Section 4), we chose to notify the user with a message that the refactoring failed because of ambiguity.

Alternative Method The algorithm above uses two-dimensional information to back-propagate code smells to AST elements and handles code as lines and columns. However, source code can be viewed as a linear sequence of characters as well. Here, a simple one-dimensional data structure and interval operations could replace the fairly complicated two-dimensional approach. Despite this mechanism being seemingly simpler, it would require different input data. Both the given inputs – the AST and PMD – work with two-dimensional data. This would mean that we would either require the source file as input or the AST and PMD must provide one-dimensional data.

- The latter requirement would be needed from both tools to replace position information with char-sequence index or store it as additional data. This new data would cause an increase both in the processing time and storage space for the tools for information that probably no one else will ever use. Still, if char-sequence index data is available as input, the one-dimensional approach is preferable.

- The former requirement would introduce an additional parameter to the algorithm; namely, the original source file. In the case where source code is given

it is possible to handle the text as one-dimensional data and map the source code elements to char-sequence indexes. However, this approach has several drawbacks. First, every search would need to read the source file, which is an i/o intensive task. This is required to calculate the end of line positions. Next, the mapping of two dimensional data to character sequence indexes would have to consider whitespace. For example, when reading a tab character from the file, the algorithm has to know it is 2, 4, 8 (or other) characters long. Both parsers could have mixed tab size settings, which would make the mapping difficult. This would also affect the two-dimensional approach, but since a line just contains only a few tabs it easier to match the source code elements in a line than in the entire file.

The reverse AST-search algorithm works only with an AST and PMD's report as input. These tools and inputs are treated as third-party from the algorithm's point-of-view. Since these inputs have two-dimensional data and source code is not available, the one-dimensional mechanism would not suffice. Nevertheless, from the point-of-view of the refactoring process, the source file is given; but we still choose the two-dimensional approach because it provides more accurate matches in real-life scenarios.

Summary The reverse AST-search algorithm enabled us to select the proper origin, which is a source code element in the AST that is the input of the refactoring transformation later on. *Thus Step 5 of Process 1 is covered.*

3.5 Executing the Refactoring

Now that we have covered each preceding step, we have all the components and we are ready to perform the refactoring. As mentioned earlier in Section 3.3, the refactoring algorithms are defined as smaller, multiple generic transformations. The type of the PMD violation determines which transformation(s) will be executed. There are some complex cases where a simple transformation will not suffice and fixing will require multiple operations.

In order to fix the *missing curly braces* (the issue in Listing 1), the transformation inserts a *block* statement into the *then* clause of the `if` statement and rewires every former member of the *then* clause to make a member of the *block* statement. See Figure 1 as an illustration of a process like this.

Once the refactoring transformation completes its operations, a new, modified, issue-free AST is created. *Thus Step 6 of Process 1 is covered.*

3.6 Generating Source Code and Creating Diffs

In the previous step, we completed the refactoring at the model level. Even though the refactoring process came to an end, there is one more thing to do. Because our main goal is to assist developers, the next task is to translate the AST back to source code where they can readily interpret the changes.

Listing 6: Refactored code

```
1 public class Example {
2     public static int limiter(int x) {
3         if (x > 10) {
4             return 10;
5         }
6         return x;
7     }
8 }
```

3.6.1 Generating Source Code

The source code generation is realized by systematically going through the AST and writing code to a text file according to the underlining source code element. For example, if we start at a file, if there is a package declaration we write the package keyword following the name of the package and a semicolon to close the statement. This is followed by import statements and so on. The generation goes on until every source code element is visited and the code is fully reconstructed from the AST.

In the case of the example in Listing 1, after the refactoring transformation we get the code shown in Listing 6. As expected, both curly braces appeared and therefore the `return` statement got a *block* around it. In consequence, the PMD rule violation got fixed, and our code maintainability improved. *Thus Step 7 of Process 1 is covered.*

3.6.2 Keeping Track of Modifications

Previously, we showed how to fully reconstruct the code from the AST. However, generating the whole code base is unnecessary. It is sufficient to recreate only those code segments where the changes occurred. There are, however, multiple ways to reduce the amount of generated code. An easy solution would be to create only those files which were affected by the refactoring. Our only concern, in this case, was that our code generation cannot reproduce exactly (100%) equivalent source code. This happens because though SourceMeter stores source code element positions and even comments, it does not store data concerning whitespaces and indentation. Despite this, we created the source code generator in such a way that it "pretty prints" the code, but what is considered "pretty" is subjective. For example, in Listing 6 the beginning bracket is positioned after the method declaration, but someone may prefer it to be on the next line. On one hand, it is possible to make this configurable. On the other hand, there are other remaining issues, such as whether there are two spaces between the *public* and the *static* keywords or whether they should be written in separate lines.

To reduce the former anomalies, we sought to minimize generation even within

files themselves. Our approach keeps track of which nodes and at what level they are modified when the refactoring operation is running. We mark those nodes that get, for instance, inserted, deleted, and swapped. Furthermore, we mark those nodes which we visited during the operation but did not modify them. For example, in Listing 6 we put a *block* statement into the *then* clause of the `if` statement and rewired the former content (the `return`) of the *then* clause as a statement of the *block* node. We marked this *block* statement as inserted, and the `return` as unmodified. The latter was required because marking a node is a recursive operation which will mark the entire subtree of that node as well. By marking the `return` statement as unmodified will leave this subtree unaffected and cause efficiency benefits.

Keeping track of modifications allowed the generator to only modify those places where it was necessary. Only the new or modified source code elements get generated, and every other part of the source code gets copied from the original source file. In the example, this works in the following way. The generator starts traversing the refactored AST from root to bottom, in a preorder strategy. While it finds unmodified nodes, it just copies the source code from the original file into the refactored file. This is based on the position information stored in the AST. This goes on until it finds a modified node in this case, an inserted *block* statement. Next, it generates the *block* statement. More precisely, it generates only the starting bracket, because there are still unvisited descendants of the *block* node in the AST. When the traversal goes to the next child, it finds an unmodified node again. It does so the same way as before, it copies the code from the original source code, only this time, it will insert the copied code with an increased indent, because the generator keeps track of the fact that we are now in a *block* statement. After every descendant has been visited and copied, we arrive back at the leave-visit for the *block* statement. The generator inserts the closing bracket into the right place, and a visit continues. Since every other node is unmodified, everything else is copied, and the generation is complete.

Generating only the required code parts created nearly the same code as the original, with most indentation and whitespaces in the right place. This was an important request from developers because interviews showed that they did not want to bother with fixing the indentation [30].

3.6.3 Creating Diffs

As soon as the generation process ended, it become possible to present the refactored code to the developers. However, reviewing entire unannotated files is not a welcomed idea by developers. Since this is the output of an automated process, users like to check what changes the automation process will apply.

To meet the need of the users, we will only show their a patch (unified diff file) as output. This patch file contains the differences between the refactored source file and the original one. This enables the developer to review what changes the automation process made on the code. Besides this, it allows the user to make a decision at the end of the process of whether to accept or reject the suggested

Listing 7: Output diff file

```
--- Example.java (original)
+++ Example.java (refactored)
@@ -1,7 +1,8 @@
 public class Example {
     public static int limiter(int x) {
-        if (x > 10)
+        if (x > 10) {
             return 10;
+        }
         return x;
     }
 }
```

refactoring. If it is the former, the user can apply the diff on the original source code, and this will transform it into the refactored code. *Thus Step 8 of Process 1 is covered.*

For example, Listing 7 shows the diff file for the refactoring of the example in Listing 1. Note that this introduces which lines are marked for deletion (starting with a "-") and which ones are marked as added (starting with a "+").

4 Evaluation

Previously, in Section 2 we mentioned that we were able to test our approach on a number of systems with the assistance of software engineering companies in an EU project. These companies were founded only in the last two decades and some of their projects were initiated before the millennium. Their projects consisted of about 5 million lines of code altogether, written mostly in Java, and covered different IT areas like ERPs (enterprise resource planning and business process management), ICMS (integrated collection management systems) and online PDF Generation. Participating companies received an extra budget to improve their source code by refactoring. First, they did the task manually, and later, after we had developed the automated tool, they used the tool to commit thousands of automatic refactorings.

At the end of the project, our tool was tested exhaustively. The companies participating in the project performed around 6,000 refactorings altogether which fixed over 11,000 coding issues. Interviews with the developers revealed that they found it to be easy to work with the tool in a lots of situations and they intend to keep using it in the future. An in-depth analysis of the results was discussed in a previous study [30] and a summary of our observations is available as an experience report [28].

5 Discussion

5.1 Performance

Our refactoring tool was implemented in Java. One of the requirements for our tool was for it to be quick-acting from a user perspective. This required that we to optimize each step for speed. The most important optimization we preformed was with the Reverse AST-search algorithm.

Building the spatial database for the whole system was an unnecessary overhead. To reduce the search space, we built the spatial index based on the issue the user had selected. The rule violation has information about which source file it is in. Only these file elements were added to the R-tree and it greatly reduced the search space. As a comparison, building the entire search space on a PC[2] for the log4j[3] project took 221 ms and used 46 MB of memory, while building only one file took 53 ms and memory used was less than a kilobyte.

Further optimizations helped to speed up the process as well. For example, the the filtering step moved a few steps ahead in the Reverse AST-search algorithm's execution order. Filtering was applied while building the search space. Only those AST nodes were added to the R-tree where the type of node matched the type of refactoring transformation parameter. This way, executing a single search operation takes less than a millisecond.

The above improvements besides some other tweaks made our tool quick and this appealed to developers. We did not make detailed measurements of the tool's performance in our studies, but in general the tool performed well.

5.2 Threats to Validity

We have identified a few validity threats that might affect the the internal and external validity of our results. Here, we discuss the validity of our findings.

Usage of Java

We have only considered Java as the target of our actions. Some of the other languages may require different approach. Nevertheless, our process is readily adaptable to most text-based programming languages.

Application of a Third-Party Tool

We provided support to the developers in identifying coding issues with a third-party static analyzer, namely PMD. Naturally, this was a great help in identifying problematic code fragments, but it might have introduced many unnecessary steps during the refactoring process. There is a risk here that by using other analyzers or by using our own, we might skip the AST-search part. For example, if we were to develop our own issue finder tool, we could directly report the AST node where the

[2]Intel i7 3.40 GHz with 8 GB ram

[3]`http://logging.apache.org/log4j/1.2`

problem is located. However, our process makes our refactoring tool independent of a single third-party static analyzer. The way, it is constructed makes it capable to switch to another analyzer with only a slight modification.

6 Related Work

Since Opdyke introduced the term *refactoring* in his PhD dissertation [20] and Fowler published a catalog of refactoring 'bad smells' [6], many researchers have studied this technique to improve the maintainability of software systems. Just a few years later, Wake [31] published a workbook on the identification of 'smells', and suggested ways of recognizing the most important ones and some possible ways to fix them by applying the appropriate refactoring techniques. Five years after the appearance of Fowler's book, Mens et al. [16] published a survey with over 100 related papers in the area of software refactoring. But the popularity of the topic continues to this day.

Automation techniques can support the regular task of refactoring and they are intensively studied by researchers. Ge et al. implemented the BeneFactor tool which detects developers' manual refactoring and reminds them that automation is available; and then it performs the refactoring automatically [9, 8]. Vakilian et al. proposed a compositional paradigm for refactoring (automate individual steps and letting programmers manually compose the steps into a complex change) and implemented a tool available that support it. Henkel et al. implemented a framework which captures and replays refactoring actions [12]. Jensen et al. used genetic programming for automated refactoring and the introduction of design patterns [13]. Also, there are many approaches to support specific refactoring techniques, like extract method [3, 26], refactoring to design patterns [1] and performing clone refactoring [33].

In a recent study, Fontana et al. examined refactoring tools designed to remove code smells [5]. They evaluated the following tools: Eclipse,[4] IntelliJ IDEA,[5] JDeodorant,[6] and RefactorIT.[7] In the case of JDeodorant, they said that this "*is the only software currently available [that is] able to provide code smell detection and then to suggest which refactoring to apply to remove the detected smells.*" To evaluate the other refactoring tools, they relied on the code smell identification of iPlasma[8] and inCode[9]. In an earlier study, Pérez et al. also identified smell detection and automatic corrections as an open challenge for the community, and proposed an automated bad smell correction technique based on the generation of refactoring plans [23]. Code-Imp is an automated refactoring platform designed for the Java language by Moghadam at el. [17]. It can apply a range of refactorings,

[4] `https://www.eclipse.org/`
[5] `http://www.jetbrains.com/idea/`
[6] `http://www.jdeodorant.com/`
[7] `http://sourceforge.net/projects/refactorit/`
[8] `http://loose.upt.ro/reengineering/research/iplasma`
[9] `http://www.intooitus.com/products/incode`

it supports several search types, and implements over 25 software quality metrics that can be combined in a variety of whys to form a fitness function.

There are many IDEs available with automatic refactoring capabilities and they support typical code restructurings (e.g. renaming variables, classes) and some common refactorings from the Fowler catalog. For instance, IntelliJ IDEA was one of the first IDEs to implement these techniques and it is able to support many languages (e.g. PHP, JavaScript, Python), not just Java which it was originally designed for. Eclipse and NetBeans[10] also implement similar algorithms. However, these IDEs have only limited support for the automatic refactoring of programming flaws. Eclipse calls these *Quick Fixes*. For example, it can organise imports, remove dead code and optimize string concatenations. NetBeans calls them *Java Hints*. These are able to fix things such as missing switch cases, suspicious equals calls or a dead branch. NetBeans also reports Java code metrics flaws, like *Class has too many methods* or *Method with multiple return points*. Moreover, it is possible to create custom, user-defined Java Hints.

The IDEs mentioned above use an incremental build system to bound text to certain source code elements. In contrast, our approach uses a reverse AST-search method to locate AST nodes. What is more, the above tools are IDEs with refactoring capabilities, while our tool is a server-side refactoring framework and its IDE plugins are simply interfaces that communicate with the server.

7 Final Remarks

In this article, we presented an automated process for refactoring coding issues. We used the output of a third-party static analyzer to find refactoring suggestions. Then we created an algorithm that is capable of locating a source code element in an AST based on textual position information. The algorithm transforms the source code into a searchable geometric space by building a spatial database. After, we conducted an exhaustive evaluation which confirmed that our approach can be adapted to a real-life scenario, and it definitely provides viable results.

In the future, we plan to use a reverse AST-search to create a mapping between multiple, different structured ASTs. Also, we would like to investigate how effectively our approach can be used to determinate what source code element the user is examining, just by looking at their text selection.

Acknowledgment

This research work was supported by the EU supported Hungarian national grant GOP-1.2.1-11-2011-0002. Here, I would also like to thank my supervisor, Rudolf Ferenc, for his support in this research work.

[10] `https://www.netbeans.org/`

References

[1] Christopoulou, Aikaterini, Giakoumakis, E. A., Zafeiris, Vassilis E., and Soukara, Vasiliki. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202–1214, November 2012.

[2] Dig, Danny and Johnson, Ralph. The Role of Refactorings in API Evolution. In *Proc. of the 21st IEEE Int. Conference on Software Maintenance (ICSM2005)*, pages 389–398. IEEE Comp. Soc., 2005.

[3] Feldthaus, Asger and Møller, Anders. Semi-automatic rename refactoring for javascript. *SIGPLAN Not.*, 48(10):323–338, October 2013.

[4] Ferenc, Rudolf, Beszédes, Árpád, Tarkiainen, Mikko, and Gyimóthy, Tibor. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, 2002.

[5] Fontana, Francesca Arcelli, Mangiacavalli, Marco, Pochiero, Domenico, and Zanoni, Marco. On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015*, XP '15 workshops, pages 7:1–7:8, New York, NY, USA, 2015. ACM.

[6] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] FrontEndART Ltd. SourceMeter website: https://www.sourcemeter.com.

[8] Ge, Xi, DuBose, Quinton L., and Murphy-Hill, Emerson. Reconciling manual and automatic refactoring. In *Proc. of the 34th Int. Conference on Software Engineering (ICSE2012)*, pages 211–221. IEEE Press, 2012.

[9] Ge, Xi and Murphy-Hill, Emerson. Benefactor: A flexible refactoring tool for eclipse. In *Proc. of the ACM Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA2011)*, pages 19–20. ACM, 2011.

[10] Guttman, Antonin. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[11] Harman, Mark and Tratt, Laurence. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1106–1113, New York, NY, USA, 2007. ACM.

[12] Henkel, Johannes and Diwan, Amer. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proc. of the 27th Int. Conference on Software Engineering (ICSE2005)*, pages 274–283. ACM, 2005.

[13] Jensen, Adam C. and Cheng, Betty H.C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO2010)*, pages 1341–1348. ACM, 2010.

[14] Kim, Miryung, Zimmermann, T., and Nagappan, N. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.

[15] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., and Theodoridis, Y. *R-Trees: Theory and Applications*. Advanced Information and Knowledge Processing. Springer London, 2010.

[16] Mens, Tom and Tourwé, Tom. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[17] Moghadam, Iman Hemati and Ó Cinnéide, Mel. Code-imp: A tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 41–44, New York, NY, USA, 2011. ACM.

[18] Mylopoulos, John, Chung, Lawrence, and Nixon, Brian. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[19] O'Keeffe, Mark and Cinnéide, Mel Ó. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502 – 516, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[20] Opdyke, William F. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[21] Oracle Corporation. OpenJDK website: http://openjdk.java.net/.

[22] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M. S. Search-based refactoring: Towards semantics preservation. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 347–356, Sept 2012.

[23] Pérez, Javier and Crespo, Yania. Perspectives on automated correction of bad smells. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 99–108, New York, NY, USA, 2009. ACM.

[24] Pinto, Gustavo H. and Kamei, Fernando. What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow. In *Proc. of the 6th Workshop on Refactoring Tools*, pages 33–36. ACM, 2013.

[25] Rigaux, Philippe, Scholl, Michel, and Voisard, Agnes. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[26] Silva, Danilo, Terra, Ricardo, and Valente, Marco Tulio. Recommending automated extract method refactorings. In *Proc. of the 22nd Int. Conference on Program Comprehension (ICPC2014)*, pages 146–156. ACM, 2014.

[27] Szőke, Gábor, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality. In *Proc. of ICCSA 2014*, pages 524–540. Springer, 2014.

[28] Szőke, Gábor, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. Designing and Developing Automated Refactoring Transformations: An Experience Report. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 693–697, March 2016.

[29] Szőke, Gábor, Nagy, Csaba, Fülöp, Lajos Jenő, Ferenc, Rudolf, and Gyimóthy, Tibor. FaultBuster: An Automatic Code Smell Refactoring Toolset. In *Proc. of SCAM 2015*, pages 253–258. IEEE, 2015.

[30] Szőke, Gábor, Nagy, Csaba, Hegedűs, Péter, Ferenc, Rudolf, and Gyimóthy, Tibor. Do Automatic Refactorings Improve Maintainability? An Industrial Case Study. In *Proc. of ICSME 2015*, pages 429–438. IEEE, 2015.

[31] Wake, William C. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., 1 edition, 2003.

[32] Xing, Zhenchang and Stroulia, Eleni. Refactoring Practice: How It is and How It Should Be Supported - An Eclipse Case Study. In *Proc. of the 22nd IEEE Int. Conference on Software Maintenance (ICSM2006)*, pages 458–468. IEEE Comp. Soc., 2006.

[33] Yoshida, Norihiro, Choi, Eunjong, and Inoue, Katsuro. Active support for clone refactoring: A perspective. In *Proc. of the 2013 ACM Workshop on Workshop on Refactoring Tools (WRT2013)*, pages 13–16. ACM, 2013.

Acta Cybernetica **23** (2017) 737–756.

Prediction of RoHCv1 and RoHCv2 Compressor Utilities for VoIP

Máté Tömösközi[ac], Patrick Seeling[b], Péter Ekler[a], and Frank H.P. Fitzek[c]

Abstract

Modern cellular networks utilising the long–term evolution (LTE) and the coming 5G set of standards face an ever–increasing demand for low–latency mobile data from connected devices. Header compression is employed to minimise the overhead for IP–based cellular network traffic, thereby decreasing the overall bandwidth usage and, subsequently, transmission delays. Here, we employ machine learning approaches for the prediction of Robust Header Compression version 1's and version 2's compression utility for VoIP transmissions, which allows the compression to dynamically adapt to varying channel conditions.

We evaluate various regression models employing r^2 and mean square error scores next to complexity (number of coefficients) based on an RTP specific training data set and separately captured live VoIP audio calls. We find that the proposed weighted Ridge regression model explains about at least 50 % of the observed results and the accuracy score may be as high as 94 % for some of the VoIP transmissions.

Keywords: robust header compression, mobile multimedia, cellular networks, bandwidth savings, linear regression, machine learning

1 Introduction

Nowadays IP–based communication is prominent in various industry applications worldwide and it is even expected to increase as indicated, e.g. in [20]. With the wide distribution of LTE enabled smart–phones and Internet capable devices (IoT), as well as the adoption the coming 5G standard, user and machine alike are constantly connected to service providers, regardless whether of they are motionless

[a]Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Hungary, E-mail: [mate.tomoskozi, ekler.peter]@aut.bme.hu

[b]Department of Computer Science, Central Michigan University, Mount Pleasant, MI 48859, USA, E-mail: pseeling@ieee.org

[c]Communication Networks Group, Technische Universität Dresden, Germany, E-mail: [mate.tomoskozi, frank.fitzek]@tu-dresden.de

DOI: 10.14232/actacyb.23.2.2017.17

or in motion. For real–time IP–based wireless connections, packets have to be sent in such a way that the users and applications experience only a minimal degradation in service quality.

During wireless communication the limited bandwidth and the relatively higher rate of errors require special attention. Since there is a large inequality between packet payloads and protocol headers during transmissions, there exists a common packetisation overhead for each packet. As an example, for a normal audio–only session the operators can save more than half of the data transmission costs by compressing various fields in the protocol headers above the link layer. This is possible because most of the fields in an IP traffic contain values that are constant or increase in a predetermined way during a transmission.

The reduction of this packetisation overhead sent over cellular and other wireless networks is typically realised through header compression mechanisms. Header compression approaches reduce the protocol encapsulation overhead between, say, wireless base stations and mobile clients. The compression approach is commonly based on a compressor/decompressor concept operating between the data link layer and the network layer on a sender/receiver pair's protocol stack implementations.

The first IP header compression scheme was the *Compressed Transport Control Protocol* (CTCP or VJHC). It was proposed by Van Jacobson [11] and it only handles the TCP protocol. CTCP combines TCP and IP headers together for better results and lower complexity. The compression algorithm itself employs *delta coding*. The advantage of this approach is the high compression ratio. Unfortunately, it is very susceptible to bit errors, which results in the dropping of many of the following packets by the receiver and there is no built–in error detection; instead it relies on the protection schemes of the lower and higher level protocols.

An improvement on this was introduced by *Perkins* in [19]. The delta coding for the neighbouring packets is replaced by a reference frame, much like modern video compressions. In this case, the first packet of a frame is sent as–is and the following packets use the delta coding to refer to the first one. This results in better tolerance to errors compared to CTCP, but it produces a lower compression gain. An improvement for the above approach by *Calveras* ([3] and [4]) proposes a dynamic frame length scheme as a function of the channel state. Both of these approaches suffer from desynchronisation when the first (uncompressed) packet is lost, which results in the corruption of all of the packets in the same frame. A suggested improvement is available from *Rossi* ([22] and [21]).

As the next step, the compression of RTP was addressed with the development of the *Compressed Real Time Protocol* (CRTP) [5]. *RObust Checksum-based COmpression* (ROCCO) [24] is a refinement of CRTP, which improves the header compression performance for highly error–prone links and long round–trip times. In a similarly way, *Enhanced Compressed RTP* (ECRTP) [6] is a refinement of CRTP.

Robust Header Compression (RoHC) builds on these predecessors and version 1 of RoHC, which was introduced in [2], and was modelled on the concept of extensibility with various profiles that were added later (see IP [12], UDP-Lite [16] and TCP profiles [18]). However, version 2 of Robust Header Compression, defined

in [17], opts for simplicity in design over extensibility.

For a brief overview of these header compression schemes see [7] or [25]. RoHCv1 was incorporated into 3GPP–UMTS and WiMAX networks with significant industry support to date and recently revised as RoHCv2 in RFC 5225 [17]. The new version increases the use–cases of RoHCv1 and it includes the enhancements proposed in–between in RFCs 3843, 4019, 4224, 4995, or 4997, without rendering any parts obsolete. The main goals of the augmented RoHCv2 are increased simplicity and robustness under similar network conditions, while maintaining or increasing performance. For a short review of Robust Header Compression's technical details, see [25].

Although both versions of RoHC achieve around 80–90 % in gain (see [28] and [26]), in spite of this neither of these compression designs accounts for the varying channel conditions that may be encountered in wireless setups (e.g., as a result of weak signals or interference). Specifically, the RFC for RoHCv2 in [17] states that "A compressor always ... repeats the same type of update until it is fairly confident that the decompressor has successfully received the information." and "The number of repetitions that is needed to obtain this confidence is normally related to the packet loss and out–of–order delivery characteristics of the link where header compression is used; it is thus not defined in this document [RFC 5525]." The impact of these repetitions is closely related to the protocol field dynamics of compressed IP–streams and it will differ from application to application.

Our present research efforts seek to address this issue by enabling current compressor implementations to configure themselves online, thereby making the compression adaptable to changing channel conditions and various network stream characteristics, which will in turn minimise compression overhead and it will pave the way for the adaptation of header compression in IoT scenarios.

Previous efforts regarding optimal configuration of the compression have only targeted the first version of RoHC. For employment with UMTS, the authors of [14] and [31] have studied and recommended optimal configurations for each compression mode to balance performance and robustness, while [15] relies on QoS information derived from the RRC layer of the radio link to dynamically configure the compression. The authors of [13] have proposed an adaptive optimistic approach, where the compressor switches to a high number of context refreshes when a negative feedback is received and uses a timer to return to the initial refresh count if no further errors occur. Again for RoHCv1, the authors of [10] use the idea of adjusting the sliding window width and the (re)initialisation timeout setting for specific (aeronautical) scenarios.

Here our current study, however, focuses on the optimal configuration of both RoHCv1 and RoHCv2 in an implementation independent and RFC compliant way which also exploits various header field dynamics to further the application of header compression for non–audio transmission use–cases as well. In [27], we investigated how the compressions react to different fluctuations in the headers and reviewed the compressor configurations which maximise utility. We showed that the appropriate choice of compressor repetition configuration increases the overall utility under dynamic channel conditions and finding the optimal configuration

might produce significant benefits. Our preliminary analysis of compression utilities and their prediction with linear regression in [29] resulted in limited, but viable prediction accuracy. Building on these we proposed a profiling scheme for RoHCv2 in [30] that provides a higher overall accuracy and we evaluated the attainable gains by employing a dynamic configuration using linear regression with polynomial basis functions.

In this study, we survey and extend our method to include version 1 of Robust Header Compression and compare the results provided by various regression methods including Support Vector Regression. These results may be employed in the RoHC compression process to dynamically adapt to changing channel conditions. Following our preliminaries, we introduce key utility concepts in Section 2. A description of the experimental setup is shown in Section 3, then in Section 4 we present our results. Finally in Section 5 we provide a brief summary and make some suggestions for future study.

2 Methods and Metrics

Below we evaluate the gains possible with dynamic configuration based on various measurements with the RoHCv1 and RoHCv2 reference implementations provided by acticom GmbH[1] for RTP traffic. We utilise systematically generated and real–life captured streams to allow consecutive evaluations with the same underlying data streams. These streams were replayed and RoHC–compressed before artificial packet losses were applied (if any). The resulting packet stream was sent to the RoHC decompressor. This is adequate since UDP does not define any feedback of its own. For the creation of our data sets we repeated this process with multiple streams and different configuration–option permutations.

Let P_i denote the payload data size of the i^{th} packet, UH_i denote the size of the i^{th} uncompressed protocol headers, and CH_i denote its compressed header size. Here we shall define actual performance measures, similar to those described in [23], as (*i*) the actual savings (or alternatively the gain) of the encapsulation overhead (headers), namely

$$SH_i = \frac{UH_i - CH_i}{UH_i}, \tag{1}$$

(*ii*) the actual savings for individual packets (with payload) as

$$SP_i = \frac{UH_i - CH_i}{UH_i + P_i}, \tag{2}$$

and (*iii*) the respective average savings for a sequence or session of N packets as

$$\overline{S} = \frac{1}{N} \sum_{i=1}^{N} S_i, \tag{3}$$

[1] See `http://www.acticom.de`

where the latter describes the bandwidth savings from a network provider point of view.

In order to accurately predict the usefulness of the compression, we need to take into account parameters that potentially decrease the overall compression gain. In this study, we define a utility function as

$$u_i(\mathbf{x}) = \{\mathbf{x} \in \mathbb{R}^k : u_i(x) \in [-1.0, 1.0] \subset \mathbb{R} \text{ and } k, i \in \mathbb{Z}\},$$

where $\mathbf{x}$ is a vector representing the k observed (independent) variable values for a given packet i. A u_i value of 1.0 is interpreted as the optimal utility and a -1.0 value as the most disadvantageous.

In our view, conditions that counterbalance optimal utility are decompressor feedback and decompression failure. During compression, feedback is produced by the decompressor and sent back to the compressor entity either on a dedicated channel or piggybacked onto a compressed packet (in the case of a peer–to–peer setup). Since compression feedback is generated by the compression layer and not as part of the original stream, each feedback byte decreases the overall gain. If there are decompression failures (commonly due to decompressor desynchronisation caused by extensive packet losses on the channel), each compressed packet failing decompression is sent completely in vain (the RoHC RFC forbids the interpretation of such packets). In this case, each byte of the compressed packet decreases the compression gain as well, since the transmission of such packets wastes bandwidth.

We will consider the following approach for the generation of the utility function. Let SH_i denote the savings of the i^{th} packet as defined in Equation 1, and FB_i denote the ratio between the size of the feedback generated after the decompression of the same packet and the length of the corresponding uncompressed header. The the utility of the i^{th} packet is

$$U_i = \sum_{i=0}^{N} (\phi(i) \cdot a \cdot SH_i - b \cdot FB_i), \tag{4}$$

where the coefficients a and b allow fine–tuning of the cost of compressed packet and feedback transmissions. Here, the indicator function $\phi(i)$ represents the decompression success of the i^{th} packet as

$$\phi(i) = \begin{cases} -1, & \text{decompression of the } i^{th} \text{ packet failed} \\ 1, & \text{otherwise.} \end{cases} \tag{5}$$

For our measurements we shall assume cost symmetry for the sending on the upstream channel of the decompressor entity and receiving on the downstream channel. This assumption also presumes that the loss of individual packets does not affect the overall utility and these cases are handled by the application layer. This consideration is fair, since we only focus on RTP compression for our model and these streams almost exclusively rely on UDP as their transport layer protocol. In practice, application examples can be found in the domains of real–time audio–video transmission, like VoIP calls and video–surveillance. Consequently, we set a

and b in Equation 4 to 1.0. This simplifies the determination of the utility function to

$$U_i' = \sum_{i=0}^{N} (\phi(i) \cdot SH_i - FB_i). \tag{6}$$

We should add that this assumption is scenario–specific and represents the best–case transmission conditions, which might not be attainable in every setup. The round–trip time in cellular networks could be significantly longer and most feedback would arrive much later than in the optimal case, which might result in consecutive decompression failures (due to decompressor desyncronisation).

3 Description of the Procedure

In order to create the input needed for the compression, we have systematically produced a number of compressed streams for our evaluation setup. Let ΔID denote the IPv4 Identification Number difference between consecutive packets, ΔTS denote the RTP Timestamp difference, with other stream characteristics denoted in a similar fashion. We will determine stream types to be used for training different header field dynamics features in our models, as described in Table 1. Note that these streams represent various field changes individually and do not necessarily model real–life audio streams. We will refer to these as training streams, which tell us how the compressor implementation reacts to changes in the input.

Table 1: Systematically generated RTP streams employed for the training of the regression models.

Ideal	Optimal input stream with ΔID = 1, ΔSN = 1, ΔTS = 1 for both IPv4 and IPv6 versions.
Marker–bit	RTP Marker–bit bursts (at the start of talkspurts), as in the ideal scenario, but the Marker–bit is set (unset) every 10 packets for both the IPv4 and IPv6 versions.
Payload type	Represents changes in the RTP Payload type after every 10 packets for both the IPv4 and IPv6 versions.
SSRC	RTP SSRC switches between two specific values after every 10 packets for both the IPv4 and IPv6 versions.
ΔID	IPv4 ID field fluctuations.
ΔTS	RTP Timestamp field fluctuations.
ΔSN	RTP Sequence Number field fluctuations.
ΔM	RTP Marker–bit field fluctuations.
ΔPT	RTP Payload Type field fluctuations.

For the streams that teach header field fluctuations, we created separate versions with 5 different fluctuation probabilities (where $P_f := 0.0, 0.15, 0.25, 0.35$ and 0.5). We also examined different discrete configurations for the building of the models, namely the optimistic mode repetition count in the interval of $[0; 5]$. Each generated stream observes the logical requirements of RTP/UDP/IP traffic, meaning that, for instance, the IP ID or the RTP Sequence Number always increases. The fluctuation probability values are based on an uncorrelated model and combined with the channel loss simulation, along with the above mentioned compressor configurations, provide the complete set of parameters needed for the generation of the training data and the configuration of the measurement runs.

In addition to the above streams, real VoIP streams were captured as well, which we used for the validation of the generated models. For this purpose, we utilised the Asterisk VoIP server[2], which connected a fixed desktop client and an Android smartphone, both using the ZoIPer VoIP client software[3]. This configuration employed the GSM 06.10 codec, which is the full–rate audio codec version that results in 33 bytes of payload. The RTP Timestamp and Sequence numbers have a constant delta of 160 and 1, respectively. The RTP Marker–bit is always set to 0, except for the first two packets. We also employed the Ekiga open source softphone software[4] for further verification as well.

During the assessment, we measured the compression output for each of the above streams under simulated loss rates $P_l \in [0.0; 0.5)$, based on correlated loss probabilities. The classical Gilbert–Elliot model was employed to simulate these correlated losses under dynamically changing channel conditions. Losses are presumed to cover both missing and corrupted packets (we will assume that the receiving device can detect bit–errors and will discard such packets). The Gilbert–Elliot model, depicted in Figure 1, is a two–state Markov–chain, where we simplify the model as follows: (i) no errors occur in the good state and (ii) no packet is conveyed successfully in the bad state.

In our measurement setup we streamline this in such a way that only two parameters are used, i.e., $p_{g,g} := 1.0 - p_{g,b}$ and $p_{b;b} := 1.0 - p_{b,g}$. We initially assume an error–free channel with $p_{g,b} := 0.0$, which we continuously increase with a delta of 0.01. The transition from the bad state to the good state mirrors this configuration, which ensures that the loss pattern will generally contain equally long bursts of packet losses and error–free transmissions. This is sufficient for our measurements, as we are only interested in ascertaining how the compression reacts to continuously repeating losses (bursts) on the channel. Therefore, an exact statistical modelling of a WLAN setup, say, like the ones described in [8] or [9], is outside of the scope of this study.

Together with 10 correlated loss sequences generated for each loss probability instance, resulting in 500 experiments for each configuration/stream combinations, a final data set of 6500 measurement runs with a total of $6,500,000$ packets was created which we preprocessed for further analysis (e.g., calculating the mean sav-

[2]See https://www.asterisk.org for details

[3]See http://www.zoiper.com for details.

[4]See http://www.ekiga.org for details.

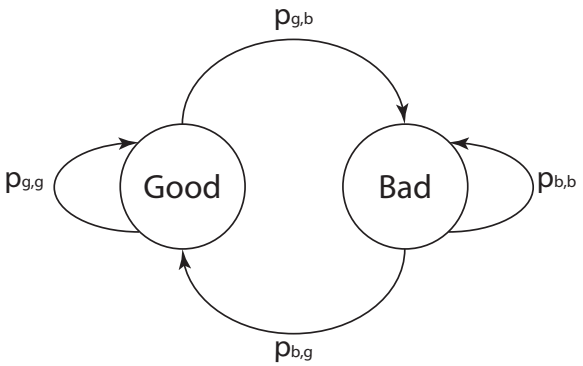


Figure 1: Gilbert–Elliot channel model with two different loss probability states g(ood) and b(ad).

ings and confidence intervals for each measurement). Here, the 6500 runs cover all the relevant *feature–configuration–loss rate* combinations. Though each run contains only 1000 packets, the model would not benefit from a longer stream length as all of the runs simulate just one type of field behaviour. This is sufficient, since if multiple behaviours were to be present in a stream, the feature with the lowest modelled utility would dominate in the output model.

As the input of the machine learning algorithms, we isolated the following independent features:

- *Measurement characteristics*: loss rate
- *Configurations*: optimistic approach repeat count
- *Stream characteristics*: $P_f(\Delta ID)$, $P_f(\Delta SN)$, $P_f(\Delta TS)$, $P_f(\Delta M)$, $P_f(\Delta PT)$, $P_f(\Delta SSRC)$ and IP version

The fluctuation probabilities (P_f) indicate the likelihood, for a given field, that its difference between two consecutive packets will diverge from the established delta value. The expected delta value is well known for certain fields (e.g., IP ID, RTP SN), but for other fields it is possible that a change occurs during the lifetime of a given transmission (e.g., RTP Timestamps). For such cases, we will assume that a constant fluctuation probability that is greater than 0.75 over ten consecutive packets indicates a change in the delta. (We should add that, in general, a fluctuation probability which is greater than 0.5 indicates a change in this delta. However, we decided to employ a greater margin of error to make the transition to a new delta less sensitive). Also, the modelling of other field fluctuations is not required, as they would be classified as *static*, *know* or *inferred*, therefore they either do not change during the lifetime of the given transmission or would belong to separate logical streams.

Overall, our modelling approach attempts to determine the utility function calculated based on Equation 6, which in turn was derived from the measured header

savings (Equation 1) as the dependent feature. To achieve this goal, we chose three different regression methods for the creation of our models and compared them with respect to their attained accuracy. Specifically, we employed Ridge regression, Bayesian linear regression, and Ridge regression with sample weights. In each case the complexity or shrinkage parameter α was set to 0.5, as this turned out to be a good choice regarding accuracy.

We decided to use Ridge regression because not all of the observed features are necessarily independent in a statistical sense or add a significant contribution to the predictor function, as seen in [29]. Ridge regression also uses the L_2 norm to avoid overfitting. We should mention, that based on our observations, a smaller regularisation term like L_1 (LASSO) or $l = 0.5$ – which is used to achieve sparsity in the independent variables – will not result in a significantly better accuracy for any of the models. Also, Bayesian regression was chosen because it avoids overfitting and it is capable of estimating the complexity parameter.

In order to achieve non–linearity in the training input, we used transformations which created polynomial combinations of the input features to the second and third order (i.e., basis functions, as in [1]). We will also provide results with Support Vector Regression (SVR) using RBF kernels as a baseline comparison. The modelling of non–linear dependencies using Neural Networks is outside of the scope of this study.

The machine learning and related techniques were implemented using the *scikit-learn*[5] data mining and data analysis library based on NumPy, SciPy and matplotlib for Python. We also refer to [1] for further details on these methods.

Since the resulting models are represented by a linear function, predictions with the final model are not computationally demanding and can readily be implemented in compressors even on resource–constrained systems.

For the weighted Ridge regression model, we employed weights as shown in Figure 2. A weight set to 100 was assigned to the ideal input stream.

This approach ensures that the model is less biased towards streams with fluctuations. We additionally assigned weights based on the observed loss rate, which was multiplied by the repeat count configuration value of the given measurement. The reason for the latter is that, based on our observations, the predictions become less accurate as the repeat count increases. By assigning higher weights this way, our model will be more reliable in general. The weight function, not taking into account stream types and configurations, is

$$w'(l) = 0.5 \cdot (l + 0.25)^3, \tag{7}$$

where $w'(l)$ denotes the weight value for a given loss rate l in $[0.0, 1.0]$. As the input weight value for the regression, we normalised w' for the $[0.0, 0.5]$ interval like so:

$$w(l) = 1.0 - \frac{w'(l)}{w'(0.5)}. \tag{8}$$

[5]See http://scikit-learn.org.

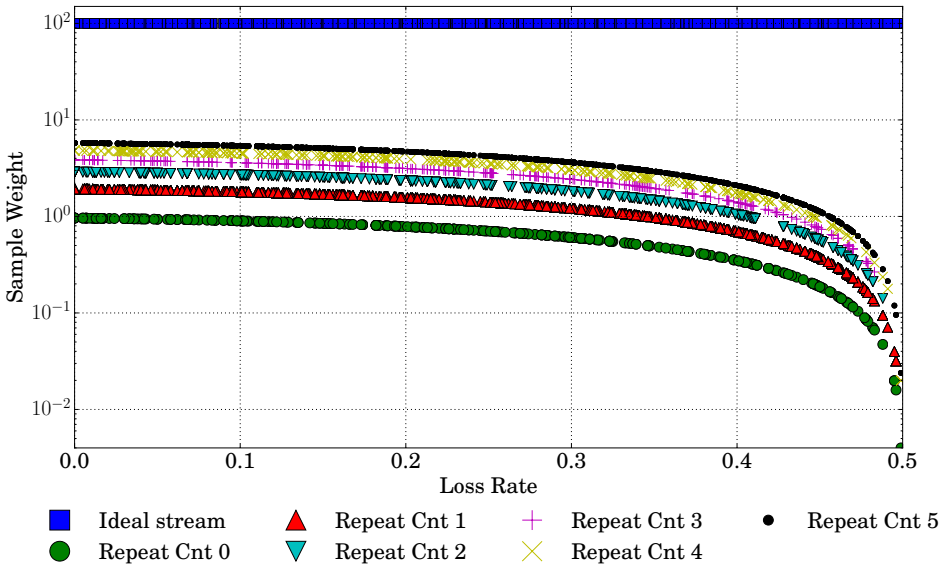


Figure 2: For the weighted Ridge–approach, sample weights are assigned to the observations based on the type of stream being used, the repeat count configuration of the compressor and the (simulated) loss rate found during the measurement stages.

This approach ensures that the model converges less rapidly towards the higher loss rates and will be more accurate for lower ones. Note that weights are only employed during the training of the models and are not part of the input of the verification dataset.

4 The Evaluation of the Prediction Quality

Next, we will make use of the coefficient of determination, denoted by r^2, for the evaluation of the accuracy scores of the models, which tells us how well the data matches a statistical model. An r^2 value of 1.0 indicates that the model perfectly fits the data, while an r^2 of 0.0 indicates that it does not fit the data at all.

We divided our observed measurement datasets into two, depending on whether the streams were the synthetic ones used during the training of the model or those generated by VoIP transmissions (denoted as *training*, *Asterisk* and *Ekiga* in our tables, respectively). The latter ones were used for independent validation. We should mention that we did not use cross–validation techniques on our input dataset (i.e., training), as we only used it to "profile" the compressor implementation. Hence the training accuracy scores serve only as a baseline comparison. Generally speaking, one is only interested in the modelling of the captured real–life streams

(i.e., the Asterisk and Ekiga streams in this study) based on the previously modelled compressor. Since both of these streams are independent of the training dataset (which was created by a packet generator tool), they can capture the accuracy of the model quite well.

4.1 Accuracy

We will first look at the overall r^2 and mean square error results got for the models trained on up to 15 % packet loss rates for both RoHCv1 and RoHCv2, as show in Table 2. We observe that for the training datasets (based on the systematically generated traffic), the Bayesian regression is the most accurate and surpasses the other models by about 10 % at most closely followed by the non–sample–weighted Ridge regression. Also, the weighted Ridge model is on average 15 % worse than the others. This is expected, since the regression takes into account some of the streams with higher precedence, thereby it will not fit as well streams that have lower sample–weights assigned. Consequently, the overall r^2 scores are lower for the weighted Ridge models applied on the training dataset. We also see similar results for the mean square error values in Table 2b.

However, for the independent VoIP streams for version 2 of RoHC the weighted Ridge approach provides the best accuracy which may be about 0.2 better in some cases. Yet, for version 1, the predictions have a very low accuracy in general and relatively high errors, which indicates that RoHCv1's utility function cannot be accurately derived in the $[0\%; 15\%)$ loss–rate range (at least for the implementation in question).

In Table 3, we provide the predictor scores for the models trained on up to 50 % packet losses. We once again observe, that for the training dataset, the Bayesian regression is the most accurate, while for the independent VoIP streams it is the weighted Ridge approach followed by the Bayesian approach. Nevertheless, we observe that all of the models lose at least $\frac{1}{3}$ of their accuracy compared to the 15 % models. However, the results for an SVR model are also provided here as a baseline comparison. In almost every case the various regression approaches outperform the SVR in accuracy by up to 40 %.

Table 3b also shows the calculated mean square error values, which measure the difference between the observed value and the estimated value. We immediately notice that Bayesian regression gives the smallest determined error values for the training input (0.022 for v1 and 0.015 for v2), while the general and the weighted Ridge regression approaches give the smallest errors for the real–world VoIP streams (0.006 for v1 and 0.009 for v2).

We attribute this finding to the construction of our measurement scenarios. Since the weighted Ridge regression is biased towards normal behaviour, it is less likely to capture the erratic fluctuations of the streams used for the training of the regression models (i.e., the unweighted Ridge and Bayesian regressions evaluated here). This makes the utility prediction of the weighted model for the VoIP model significantly more accurate.

Table 2: Utility model accuracy up to 15 % loss–rates.

Stream	*Order*	*Ridge*		*Bayesian*		*Weighted*		*SVR*	
RoHC Version		v1	v2	v1	v2	v1	v2	v1	v2
Training	First	*0.61*	*0.69*	**0.61**	**0.7**	0.46	0.63	0.69	0.72
	Second	*0.8*	*0.8*	**0.86**	**0.87**	0.72	0.72		
	Third	*0.88*	*0.87*	**0.93**	**0.94**	0.8	0.8		
Asterisk	First	*0.4*	*0.68*	**0.41**	0.66	0.39	**0.7**	0.4	0.29
	Second	**0.43**	0.57	*0.43*	*0.59*	0.4	**0.73**		
	Third	*0.35*	0.52	0.28	*0.69*	**0.4**	**0.73**		
Ekiga	First	*0.23*	*0.84*	**0.23**	0.81	0.2	**0.93**	0.35	0.29
	Second	**0.29**	*0.64*	*0.23*	0.66	0.2	**0.94**		
	Third	**0.29**	0.56	0.08	*0.86*	*0.18*	**0.94**		

(a) r^2 values

Stream	*Order*	*Ridge*		*Bayesian*		*Weighted*		*SVR*	
RoHC Version		v1	v2	v1	v2	v1	v2	v1	v2
Training	First	*.062*	*.027*	**.061**	**.027**	.1	.036	.05	.025
	Second	*.032*	*.019*	**.023**	**.012**	.046	.026		
	Third	*.02*	*.011*	**.011**	**.006**	.032	0.018		
Asterisk	First	.023	.016	*.022*	*.014*	**.01**	**.006**	.009	.019
	Second	*.01*	.012	**.006**	*.007*	.01	**.005**		
	Third	**.007**	.009	*.008*	*.006*	.01	**.005**		
Ekiga	First	**.05**	**.008**	*.055*	*.01*	.125	.037	.049	.023
	Second	**.059**	**.017**	*.095*	.037	.125	*.035*		
	Third	**.082**	**.027**	*.124*	.038	.126	*.036*		

(b) Mean Square Error values

4.2 Complexity

We now turn to the question of the overall complexity required for solving the predictor function of the models based on the number of features and summations in the resulting polynomial equations, as suggested in Table 4.

We initially observe that as the order of the solution increases, the coefficient count of our model exponentially increases as well. However, if only coefficients which are larger than 0.01 are taken into account, the feature count may be significantly reduced by about 50 %.

Figure 3 shows the relationship between the order of the polynomial basis functions and the r^2 values relative to the measured loss rates for the unweighed Ridge regression, for both RoHCv1 and RoHCv2.

Table 3: Utility model accuracy up to 50 % loss–rates.

Stream	*Order*	*Ridge*		*Bayesian*		*Weighted*		*SVR*	
RoHC Version		v1	v2	v1	v2	v1	v2	v1	v2
Training	First	*0.66*	*0.61*	**0.66**	**0.61**	0.6	0.55	0.75	0.72
	Second	*0.85*	*0.8*	**0.87**	**0.84**	0.82	0.75		
	Third	*0.9*	*0.87*	**0.92**	**0.9**	0.87	0.84		
Asterisk	First	**0.66**	**0.24**	*0.66*	*0.24*	0.59	0.20	0.63	0.35
	Second	0.58	0.42	*0.59*	*0.43*	**0.62**	**0.45**		
	Third	0.63	0.39	*0.65*	*0.4*	**0.65**	**0.45**		
Ekiga	First	**0.49**	*0.29*	*0.49*	0.29	0.49	**0.51**	0.22	0.36
	Second	*0.48*	0.53	0.48	*0.57*	**0.59**	**0.69**		
	Third	0.55	0.5	*0.57*	*0.6*	**0.64**	**0.71**		

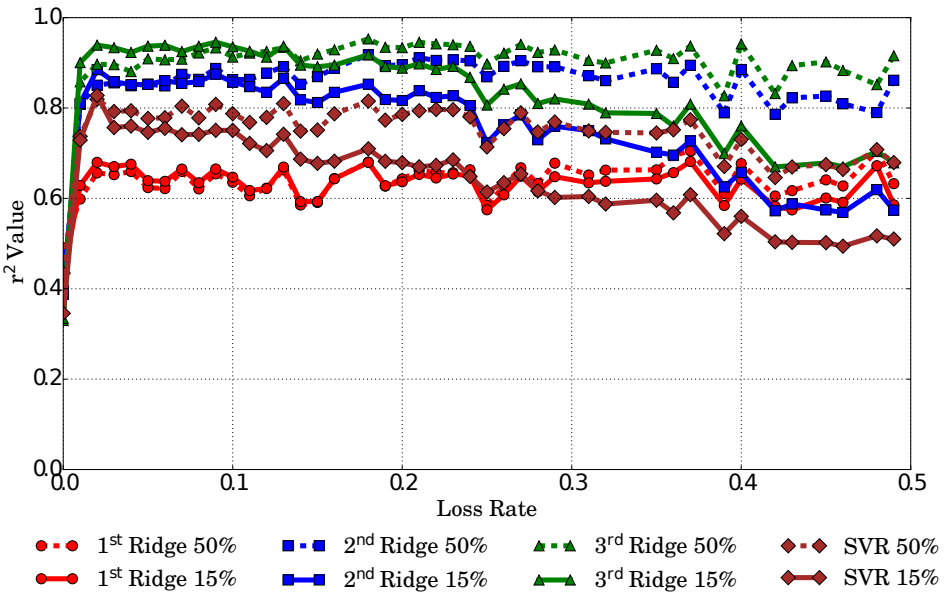
(a) r^2 values

Stream	*Order*	*Ridge*		*Bayesian*		*Weighted*		*SVR*	
RoHC Version		v1	v2	v1	v2	v1	v2	v1	v2
Training	First	*.06*	*.037*	**.06**	**.037**	.08	.046	.044	.027
	Second	*.027*	*.019*	**.023**	**.015**	.032	.024		
	Third	*.019*	.012	**.014**	**.009**	.023	.015		
Asterisk	First	*.028*	.03	**.027**	*.029*	.035	**.019**	.026	.025
	Second	**.027**	.018	*.027*	*.014*	.033	**.013**		
	Third	**.025**	*.015*	*.029*	.015	.031	**.013**		
Ekiga	First	**.039**	**.01**	*.041*	*.012*	.108	.032	.048	.015
	Second	**.057**	**.017**	*.082*	*.031*	.111	.034		
	Third	**.078**	**.029**	*.104*	.039	.111	*.035*		

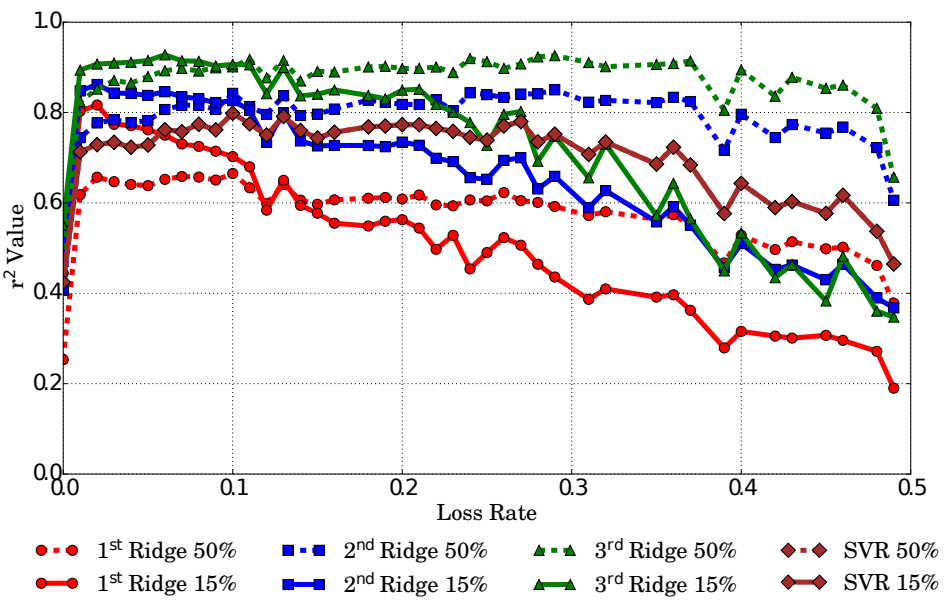
(b) Mean Square Error values

Table 4: The complexity of the models based on the independent feature counts used with the total number of summations (in parentheses) in the predictor equation and the number of features (summations) that have coefficients greater than 0.01.

Order	*All coefficients*	*Coeff's > 0.01*
First	9 (9)	9 (9)
Second	90 (99)	57 (66)
Third	495 (594)	192 (249)



(a) RoHCv1



(b) RoHCv2

Figure 3: 1^{st}, 2^{nd} and 3^{rd} order Ridge regression and SVR accuracy scores got for the r^2 values relative to the loss–rate.

We observe that while the third order solution gives the best results, the accuracy of the second order one is also quite close to that of the third order one. In general, the second order solution results in an increase of r^2 by about 0.1–0.2 compared to the first order solution, whereas the third order r^2 is about 5 % better than the second order solution in some cases. A fourth order solution, however, does not bring about any improvement in the accuracy of the model.

4.3 Accuracy and Losses

We shall now evaluate the model accuracies for the prediction of the real–life VoIP calls for different loss rates. Figure 4a illustrates the relation between the r^2 value and the loss rates of the Asterisk VoIP downstream for RoHCv2.

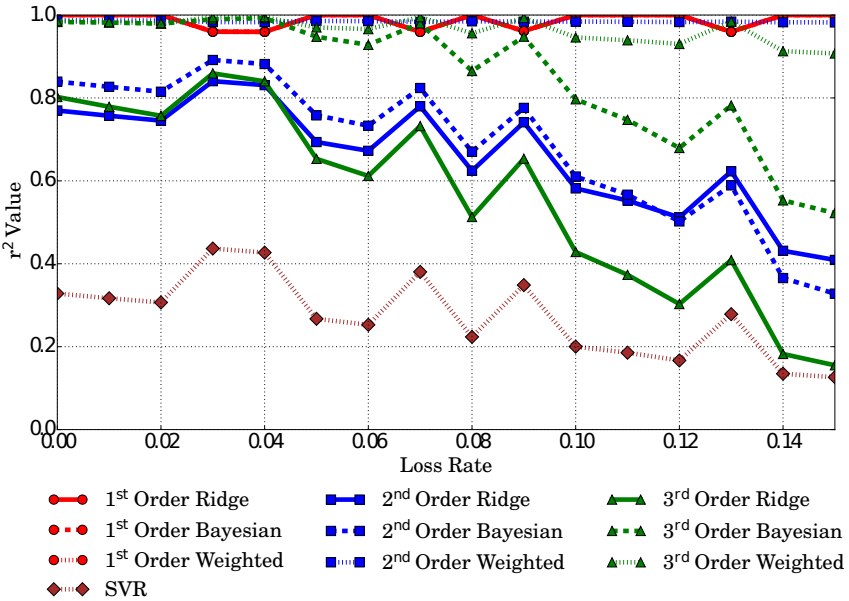
At first glance we notice that all of the first order predictors have a very high accuracy throughout. This is most likely due to this particular stream being well behaved, as in the original stream no significant fluctuations are present. For the downstream Asterisk call (not shown in the figure), the utility function starts to fluctuate quite rapidly from 8 % losses. We also see a similar trend for the Ekiga stream compression (Figure 4b).

For most of the other streams with an increase in the loss rate, the prediction accuracy falls quite rapidly, but in each case the weighted approach proves to be more stable and the accuracy score is about 90 % on average.

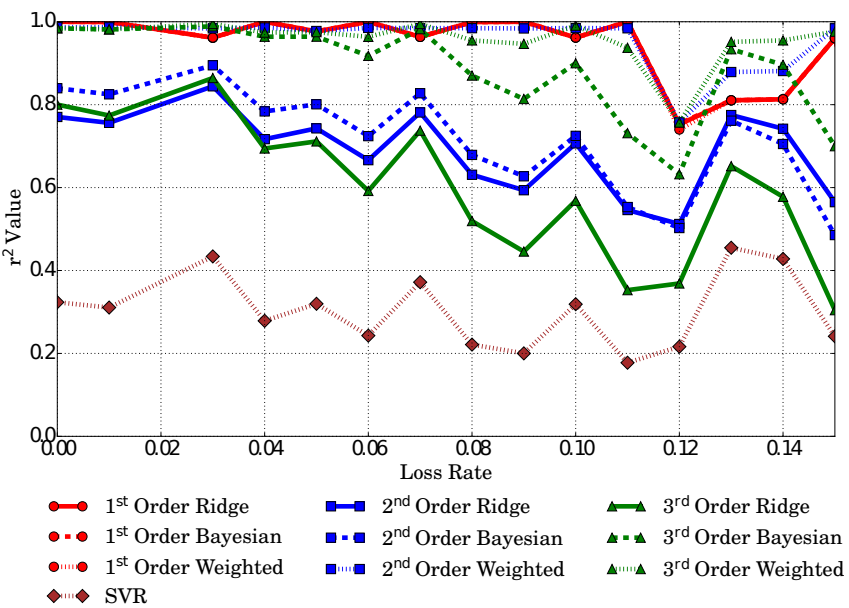
As the order of models increases, the prediction accuracy increases as well. In general, the higher order models provide a more stable prediction and the different model types remain close to each other with an increase in the order. A counter–intuitive result here is that the first order models perform better for higher loss probabilities than the more complex solutions (except for the weighted Ridge approach). For version 1 of RoHC, predicting the utility function is less reliable as the accuracy varies between 0.2 and 0.9 without any recognisable pattern. However, the r^2 scores do become unstable for losses 12 % and higher with both the Asterisk and the Ekiga streams.

5 Conclusions

Here we present a procedure that can be used for profiling Robust Header Compression (version 1 and version 2) compressor implementations. We compared Ridge and Bayesian regressions, Support Vector Regression, as well as Ridge regression with sample weights to construct a predictor in order to derive the compression utility values for RoHCv1 and RoHCv2. This will allow compressors to dynamically adapt to varying channel conditions in the future.



(a) Asterisk



(b) Ekiga

Figure 4: VoIP downstream prediction accuracy with up to 15 % losses for RoHCv2.

Based on the performance evaluation of the applied prediction, we find that the Bayesian approach predicts both the training dataset and the independent VoIP scenarios quite well and it has an r^2 value of 0.66–0.93 and 0.61–0.94 for RoHCv1 and RoHCv2 training streams, respectively. We also reviewed prediction accuracies for real–life VoIP transmissions, which yielded r^2 values of 0.5 on average for both versions.

However, in order to make the predictor less sensitive to a subset of the training streams and also more accurate for the lower loss rates, we devised a weighting strategy for Ridge regression, which is more precise for less error prone channels. This approach achieves a prediction accuracy of at least 0.5 on either of the VoIP transmissions and it may be as good as 0.94 in some cases.

We also find that the second order polynomial base functions provide a good compromise between accuracy and complexity. A second order predictor would require 99 summations, while a third order predictor needs 594. The significantly reduced number of prediction variables and the linear regression make this a suitable approach for low computational resource environments with desired high throughput. When encountering significant channel impairments or larger numbers of streams, however, the reduced prediction accuracy will only allow for a general approximation of the utility level. This issue could be mitigated with a larger and more diverse set of profiles as well as more intricate and computationally demanding approaches to the prediction, which is part of our ongoing work.

Acknowledgment

The authors thank acticom GmbH, especially Gerrit Schulte, for the support and software reference implementations of RoHC as well as their help in conducting the experiments. This work was supported by the János Bolyai Research Fellowship of the Hungarian Academy of Sciences.

References

[1] Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] C. Bormann, et. al. RObust Header Compression: ROHC: Framework and four profiles: RTP, UDP, ESP, and uncompressed. *Request for Comments 3095*, 2001.

[3] Calveras, A., Arnau, M., and Paradells, J. An Improvement of TCP/IP Header Compression Algorithm for Wireless Links. *Third World Multiconference on Systemics, Cybernetics and Informatics (SCI99) and the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS99), IEEE, Orlando, USA*, vol. 4:pp. 39–46, July/August 1999.

[4] Calveras, A. and Paradells, J. TCP/IP Over Wireless Links: Performance Evaluation. *48th Annual Vehicular Technology Conference VTC 98 IEEE, Ottawa (Ontario), Canada*, vol. 3:pp. 1755–1759, May 1998.

[5] Casner, S. and Jacobson, V. Compressing IP/UDP/RTP Headers for Low-Speed Serial Links. *Request for Comments 2508*, 1999.

[6] Chen, W.-T., Chuang, D.-W., and H.-C.Hsiao. Enhancing CRTP by Retransmission for Wireless Networks. *Proceedings of the Tenth International Conference on Computer Communications and Networks*, pages pp. 426–431, 2001.

[7] Fitzek, F. H., Hendrata, S., Seeling, P., and Reisslein, M. Header Compression Schemes for Wireless Internet Access. *Electrical Engineering & Applied Signal Processing. CRC Press*, page Ch. 10, 2004.

[8] Hartwell, J. A. and Fapojuwo, A. O. Modeling and characterization of frame loss process in IEEE 802.11 wireless local area networks. In *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, volume 6, pages 4481–4485 Vol. 6, Sept 2004.

[9] Hasslinger, G. and Hohlfeld, O. The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet. In *14th GI/ITG Conference - Measurement, Modelling and Evalutation of Computer and Communication Systems*, pages 1–15, March 2008.

[10] Hermenier, R. and Kissling, C. Optimization of robust header compression for aeronautical communication. In *Integrated Communications, Navigation and Surveillance Conference (ICNS), 2013*, pages 1–11, April 2013.

[11] Jacobson, V. *Compressing TCP/IP Headers for Low-Speed Serial Links*, February 1990. RFC1144.

[12] Jonsson, L-E. and Pelletier, G. *RObust Header Compression (ROHC): A Compression Profile for IP*, June 2004. RFC3843.

[13] Kim, J., Woo, H., Lee, H., and Lee, M. Dynamic adjustment of optimistic parameter of ROHC for performance improvement. In *2009 International Conference on Information Networking*, pages 1–3, Jan 2009.

[14] Minaburo, A., Nuaymi, L., Singh, Kamal Deep, and Toutain, L. Configuration and analysis of robust header compression in UMTS. In *Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. 14th IEEE Proceedings on*, volume 3, pages 2402–2406 vol.3, Sept 2003.

[15] Minaburo, A. C., Singh, K. D., Toutain, L., and Nuaymi, L. Proposed behavior for robust header compression over a radio link. In *Communications, 2004 IEEE International Conference on*, volume 7, pages 4222–4226 Vol.7, June 2004.

[16] Pelletier, G. *RObust Header Compression (ROHC): Profiles for User Datagram Protocol (UDP) Lite*, April 2005. RFC4019.

[17] Pelletier, G. and Sandlund, K. RObust Header Compression Version 2 (ROHCv2): Profiles for RTP, UDP, IP, ESP and UDP-Lite. *Request for Comments 5225*, 1997.

[18] Pelletier, G., Sandlund, K., Jonsson, L-E., and West, M. *RObust Header Compression (ROHC): Profiles for User Datagram Protocol (UDP) Lite*, July 2007. RFC4996.

[19] Perkins, S. J. and Mutka, M. W. Dependency Removal for Transport Protocol Header Compression over Noisy Channels. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 1025–1029, Montreal, Canada, 1997.

[20] Ribičre, M. and Charlton, P. Cisco visual networking index: Global mobile data traffic forecast update. Cisco, Inc., 2014–2019. [Online]. Available: http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html (current Feb. 2015.).

[21] Rossi, M., Giovanardi, A., Zorzi, M., and Mazzini, G. Improved header compression for TCP/IP over wireless links. *Electronics Letters*, vol. 36(no. 23):pp. 1958–1960, November 2000.

[22] Rossi, M., Giovanardi, A., Zorzi, M., and Mazzini, G. TCP/IP Header Compression: Proposal and Performance Investigation on a WCDMA Air Interface. *12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, IEEE*, vol. 1:pp. A–78 – A–82, September 2001.

[23] Seeling, P., Reisslein, M., Fitzek, F.H.P., and Hendrata, S. Video quality evaluation for wireless transmission with robust header compression. In *Information, Communications and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*, volume 3, pages 1346–1350 vol.3, Dec 2003.

[24] Svanbro, K., Hannu, H., Jonsson, L.-E., and Degermark, M. Wireless Real-time IP Services Enabled by Header Compression. *Proceedings of the IEEE Vehicular Technology Conference (VTC), Tokyo, Japan*, vol. 2:pp. 1150–1154, 2000.

[25] Tömösközi, M. Robust Header Compression in Wireless Networks. TDK Scientific Students' Association Conference at Budapest University of Technology, 2013. [Online]. Available: http://tdk.bme.hu/VIK/DownloadPaper/Robust-Header-Compression-vezetek-nelkuli (current Aug. 2016.).

[26] Tömösközi, M., Seeling, P., Ekler, P., and Fitzek, F. H. P. Performance Evaluation and Implementation of IP and Robust Header Compression Schemes for TCP and UDP Traffic in the Wireless Context. In *Engineering of Computer Based Systems (ECBS-EERC), 2015 4th Eastern European Regional Conference on the*, pages 45–50, Aug 2015.

[27] Tömösközi, M., Seeling, P., Ekler, P., and Fitzek, F. H. P. Efficiency Gain for RoHC Compressor Implementations with Dynamic Configuration. In *VTC2016-Fall Workshop on Cellular Internet of Things - Emerging Trends and Enabling Technologies*, Sept 2016.

[28] Tömösközi, M., Seeling, P., and Fitzek, F. H. Performance evaluation and comparison of RObust Header Compression (ROHC) ROHCv1 and ROHCv2 for multimedia delivery. In *Workshops Proceedings of the Global Communications Conference, GLOBECOM*, pages 1346–1350, Atlanta, GA, USA, 2013.

[29] Tömösközi, Máté, Seeling, Patrick, Ekler, Péter, and Fitzek, Frank H.P. Performance Prediction of Robust Header Compression version 2 for RTP Audio Streaming Using Linear Regression. In *European Wireless 2016 (EW2016)*, Oulu, Finland, May 2016.

[30] Tömösközi, Máté, Seeling, Patrick, Ekler, Péter, and Fitzek, Frank H.P. Regression Model Building and Efficiency Prediction of RoHCv2 Compressor Implementations for VoIP. In *2016 IEEE Global Communications Conference: Mobile and Wireless Networks (Globecom2016 MWN)*, Washington, USA, December 2016.

[31] Wang, B., Schwefel, H. P., Chua, K. C., Kutka, R., and Schmidt, C. On implementation and improvement of robust header compression in UMTS. In *Personal, Indoor and Mobile Radio Communications, 2002. The 13th IEEE International Symposium on*, volume 3, pages 1151–1155 vol.3, Sept 2002.

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János